Practical Aspects of SAT Solving

SMT'12 / PAAR'12

Affiliated to IJCAR'12

1st July 2012

University of Manchester

Manchester, UK

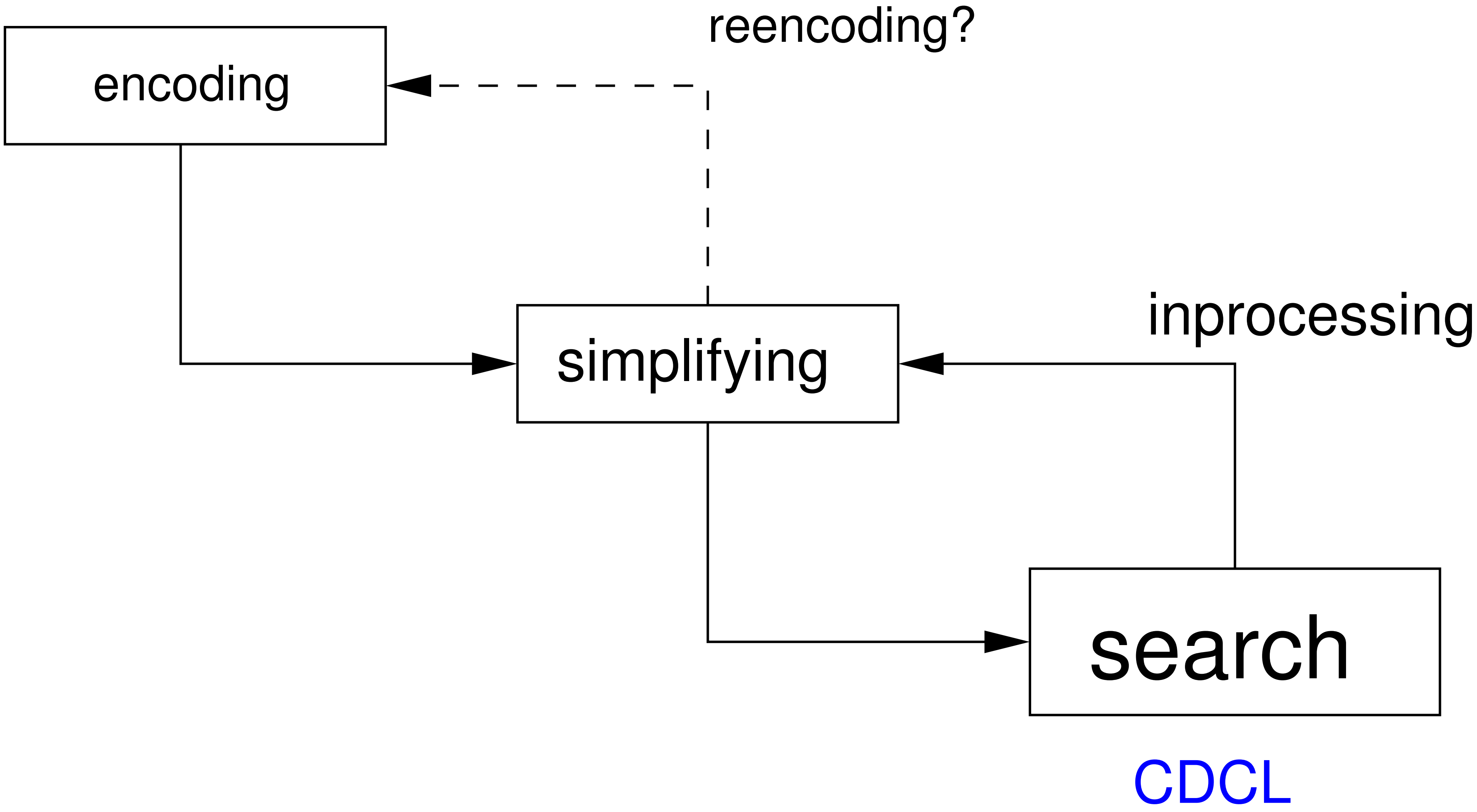
Armin Biere

Institute for Formal Models and Verification

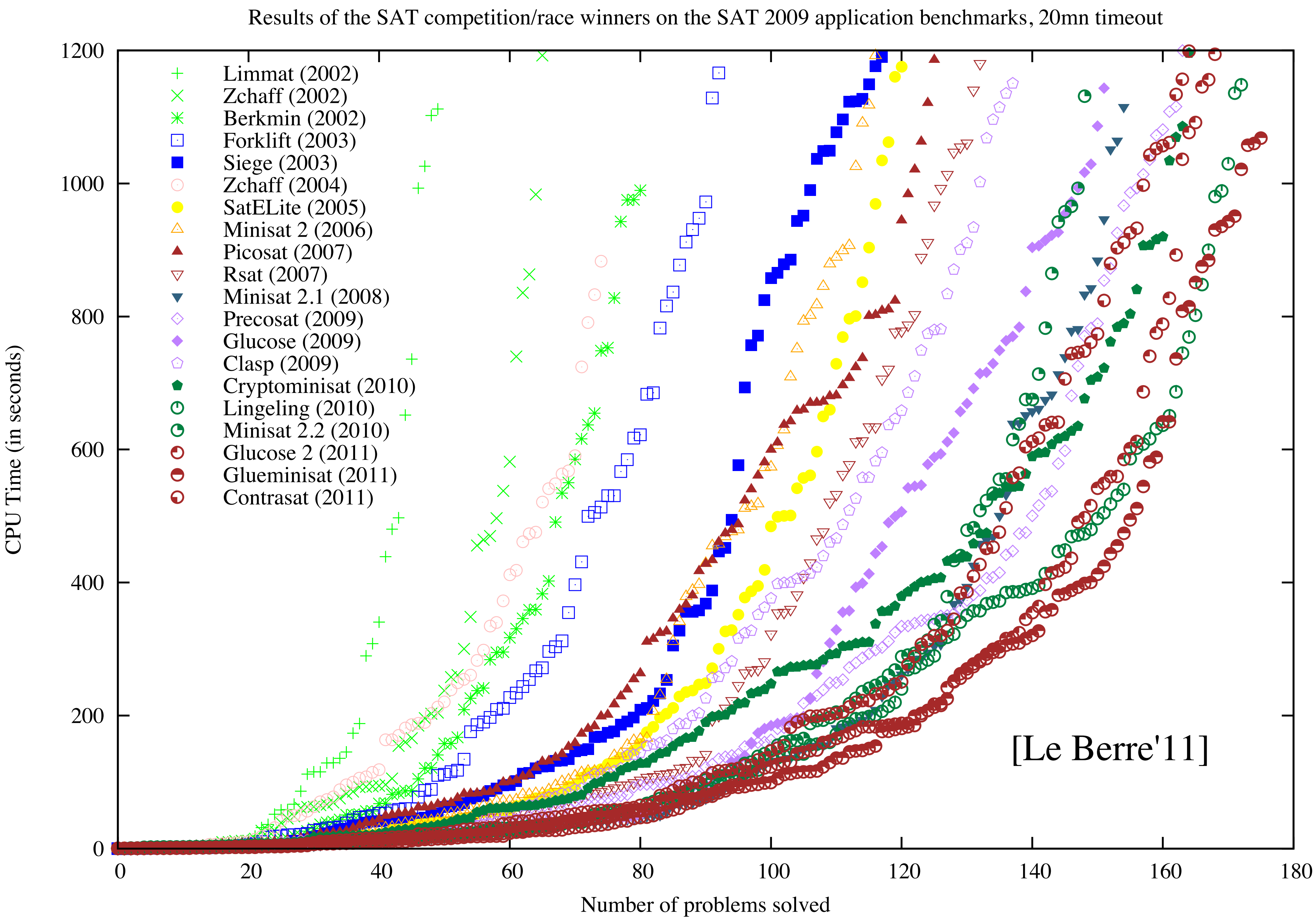
Johannes Kepler University, Linz, Austria

http://fmv.jku.at/biere/talks/Biere-SMT12PAAR12-invited-talk.pdf

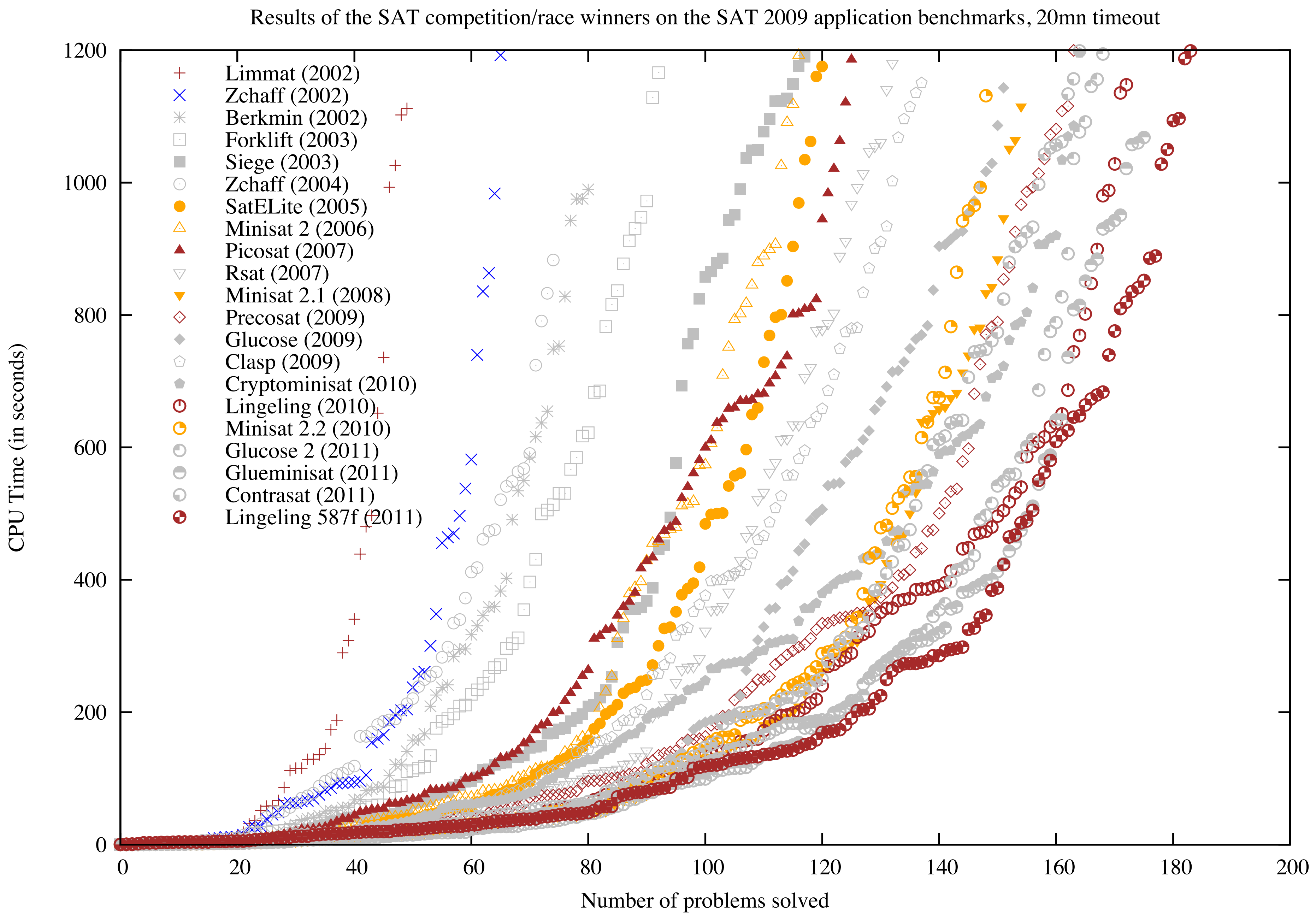
http://fmv.jku.at/cleaneling/cleaneling00a.zip


reencoding?
encoding
inprocessing
simplifying
search
CDCL

SAT Competition / Race Winners on SC 2009 Application Benchmarks



ZChaff, MiniSAT, My Solvers



DP / DPLL

- dates back to the 50'ies:

  1^{st} version DP is *resolution based* $\Rightarrow$ SatELite preprocessor [EénBiere05]

  2^{st} version D(P)LL splits space for time $\Rightarrow$ $\boxed{\text{CDCL}}$

- **ideas:**

  – 1st version: eliminate the two cases of assigning a variable in space or

  – 2nd version: case analysis in time, e.g. try $x = 0, 1$ in turn and recurse

- most successful SAT solvers are based on variant (CDCL) of the second version

  works for very large instances

- recent ($\leq$ 15 years) optimizations:

  backjumping, learning, UIPs, dynamic splitting heuristics, fast data structures

  (we will have a look at each of them)

DP Procedure

[DavisPutnam'61]

forever

if $F = \top$ **return** *satisfiable*

if $\bot \in F$ **return** *unsatisfiable*

pick remaining variable x

add all resolvents on x

remove all clauses with x and $\neg x$

$\Rightarrow$ SatELite preprocessor [EénBiere05]

D(P)LL Procedure

[DavisLogemannLoveland'62]

$DPLL(F)$

$F := BCP(F)$ boolean constraint propagation

if $F = \top$ **return** *satisfiable*

if $\bot \in F$ **return** *unsatisfiable*

pick remaining variable x and literal $l \in \{x, \neg x\}$

if $DPLL(F \wedge \{l\})$ returns *satisfiable* **return** *satisfiable*

return $DPLL(F \wedge \{\neg l\})$

$\Rightarrow$ CDCL

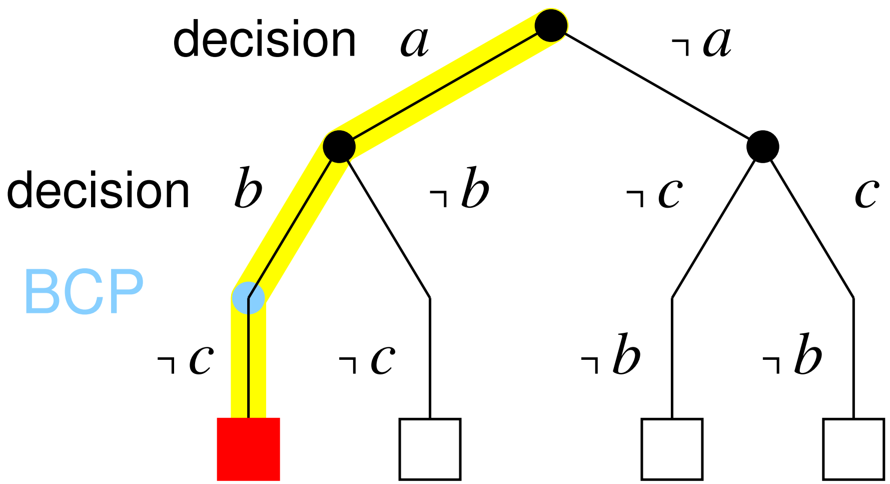
DPLL Example

[DavisLogemannLoveland'62]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

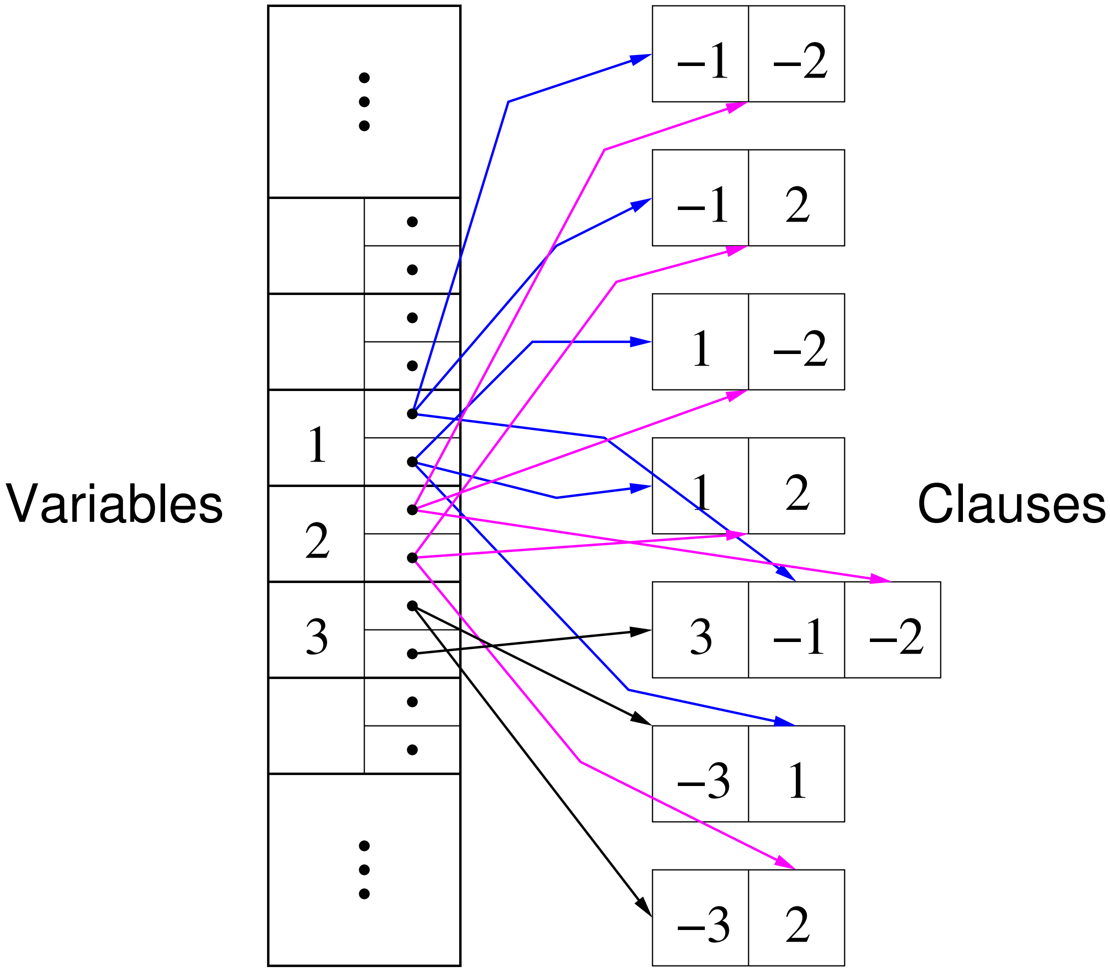
$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

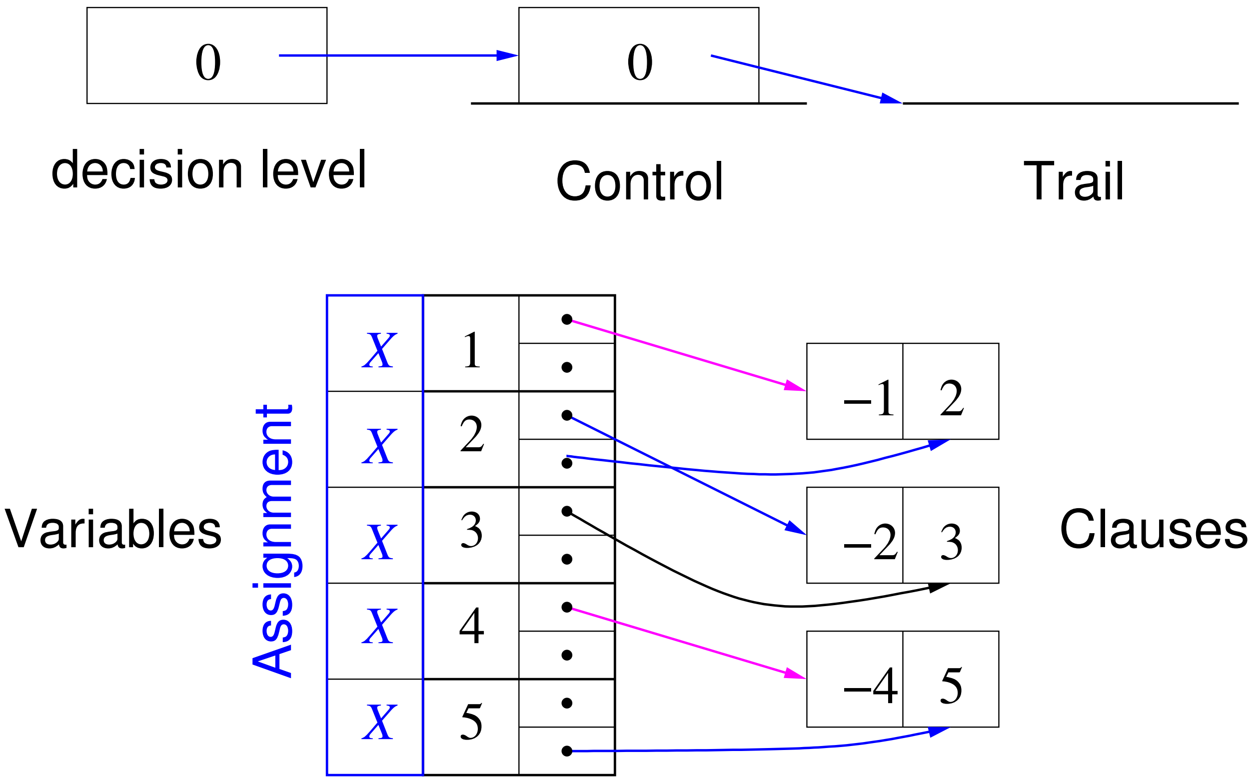
Simple Data Structures in DPLL Implementation

[DavisLogemannLoveland'62]



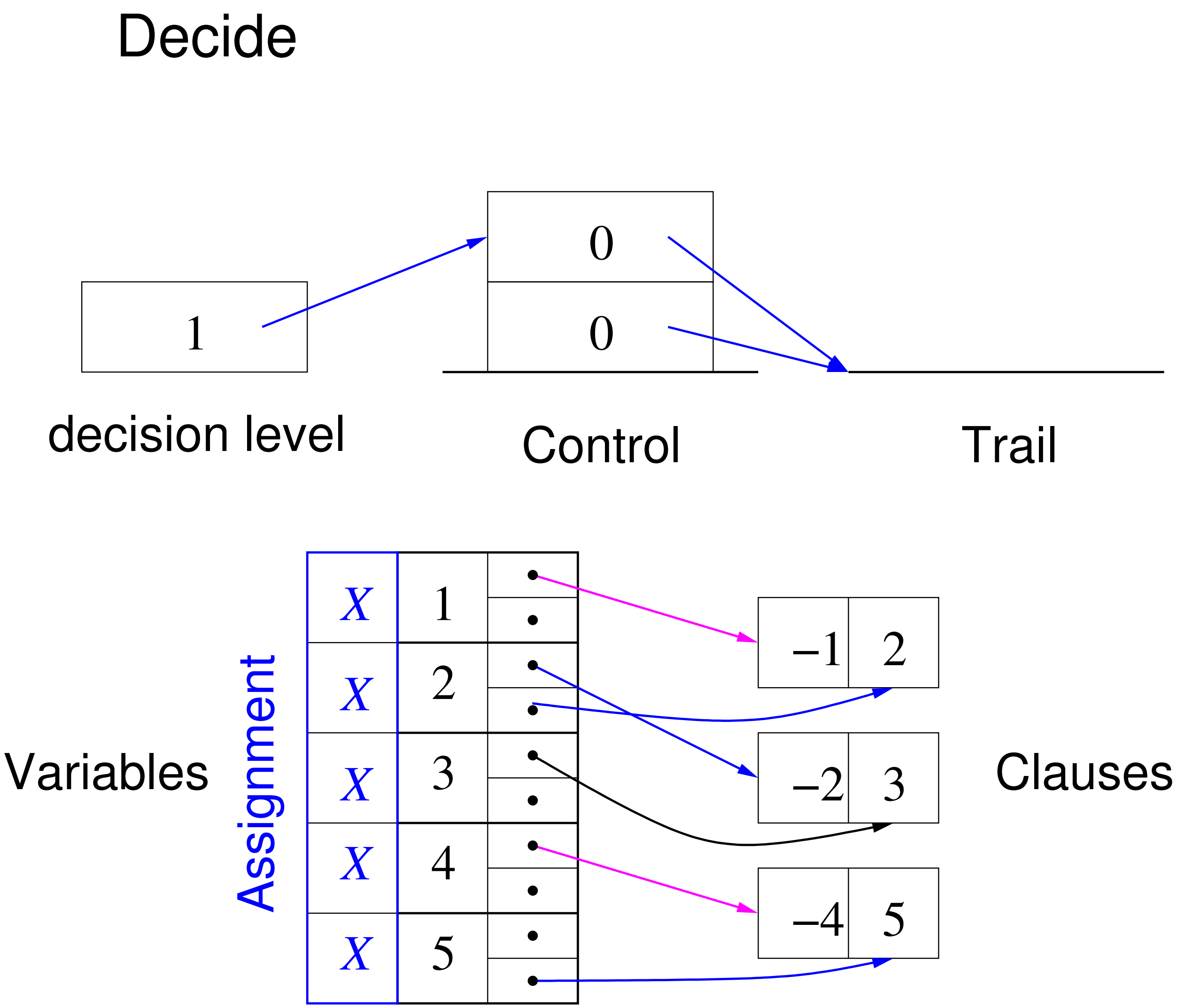
BCP Example

[DavisLogemannLoveland'62]



Example cont.

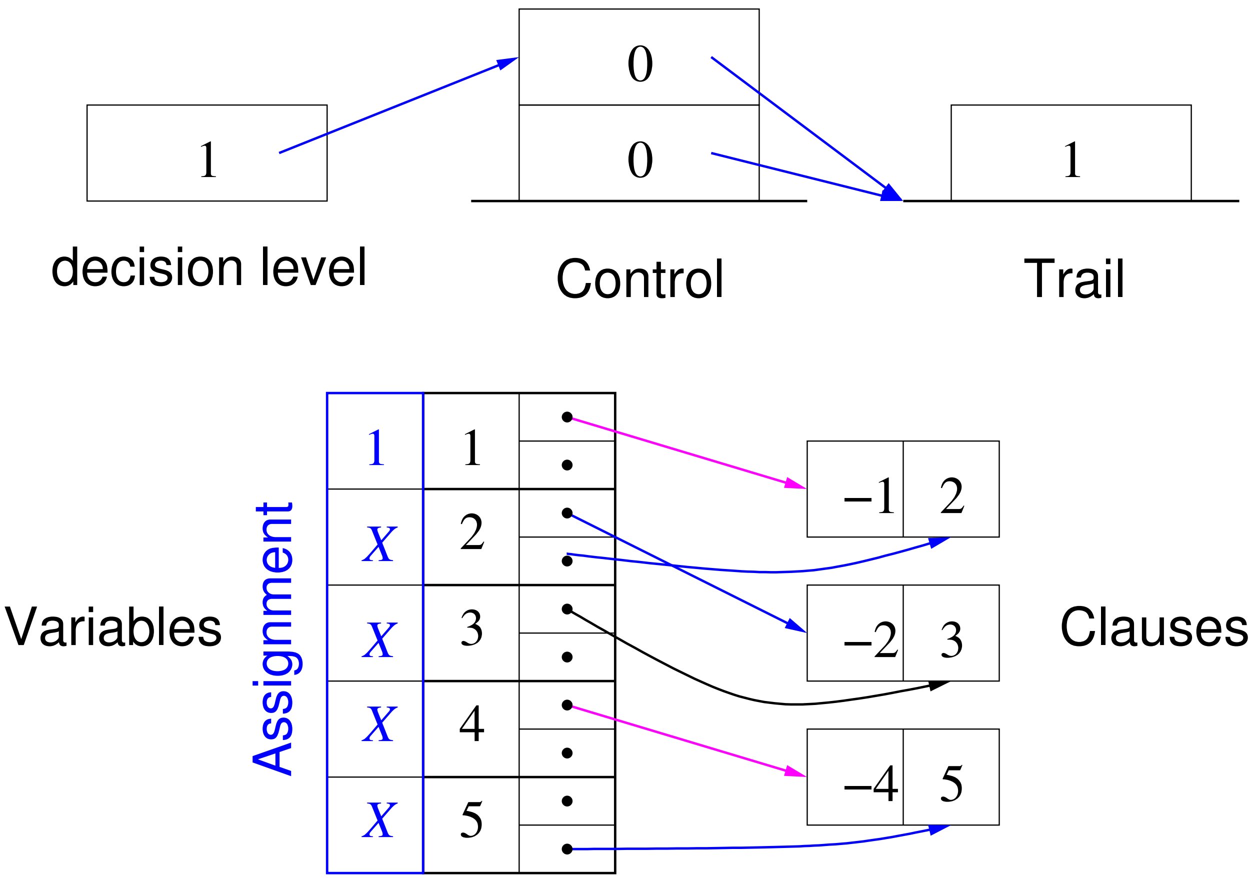
[DavisLogemannLoveland'62]



Example cont.

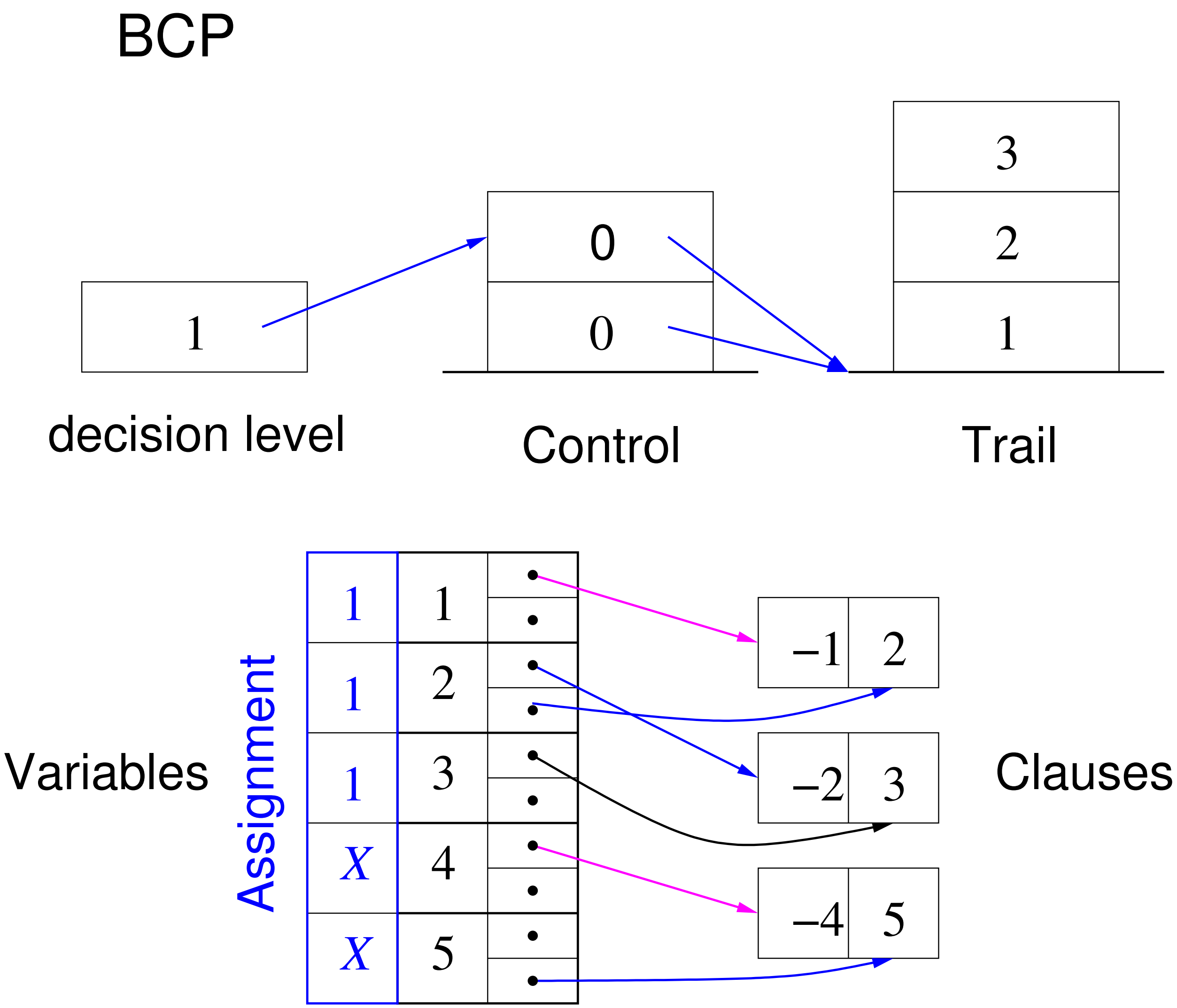
[DavisLogemannLoveland'62]

Assign



Example cont.

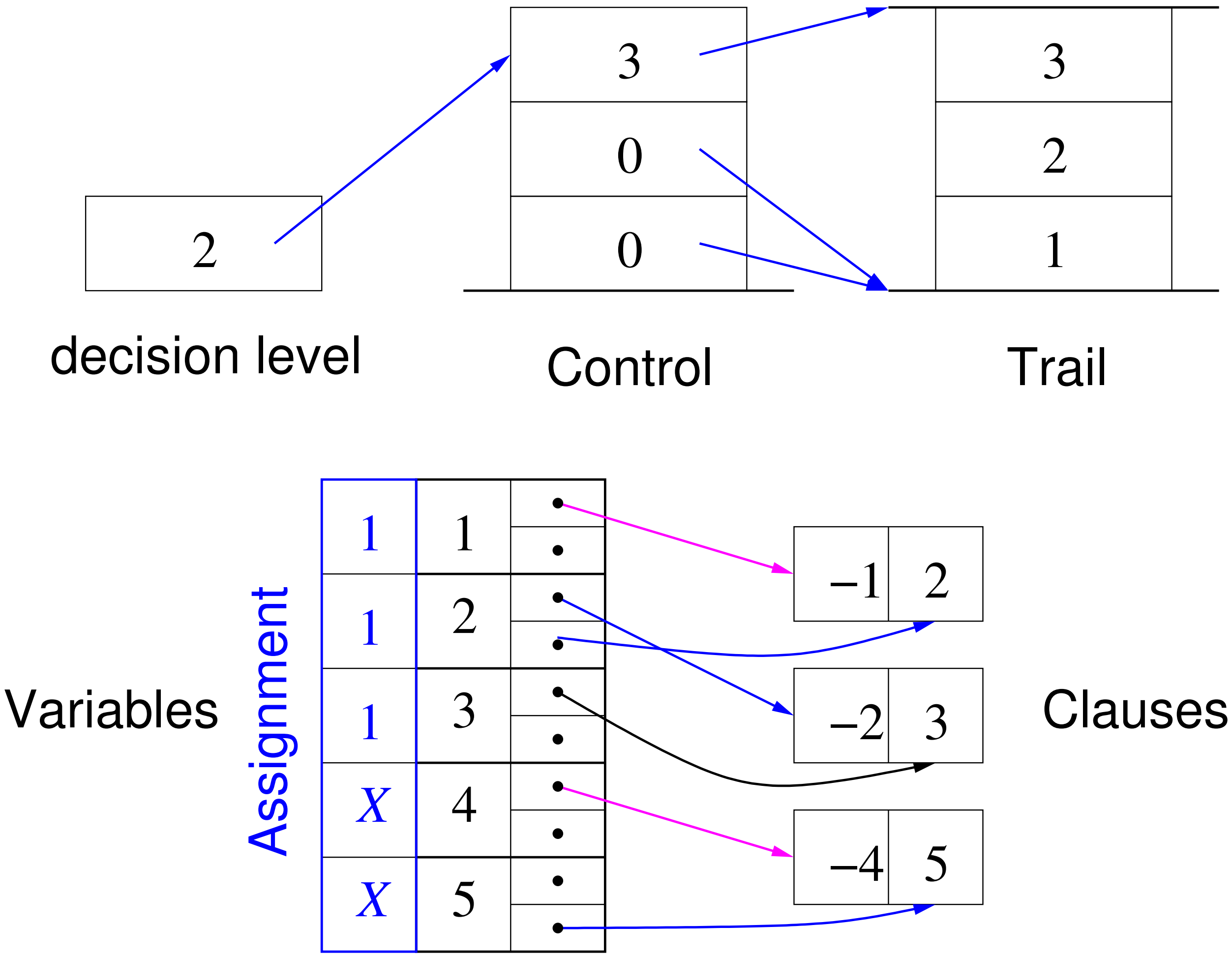
[DavisLogemannLoveland'62]



Example cont.

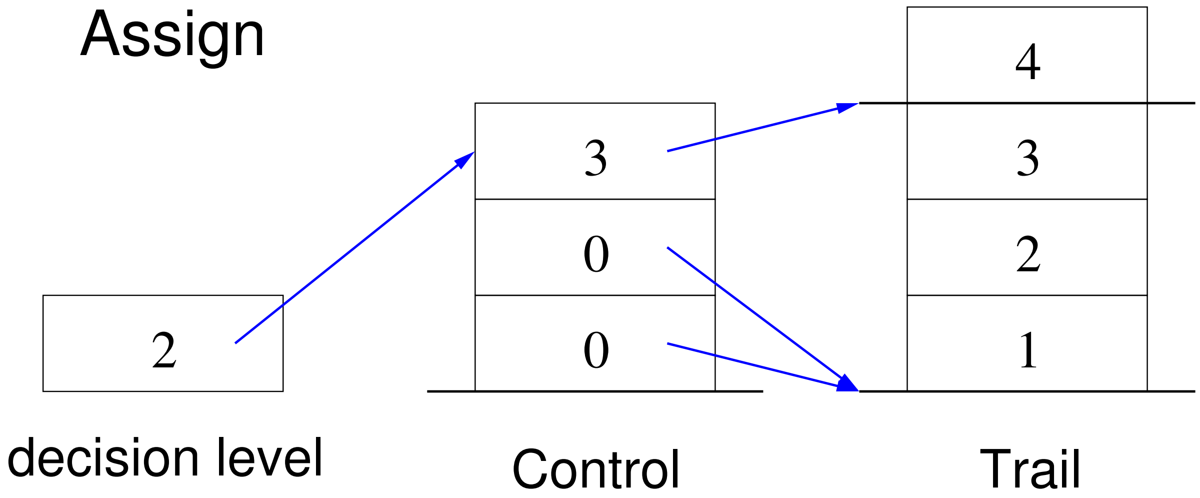
[DavisLogemannLoveland'62]

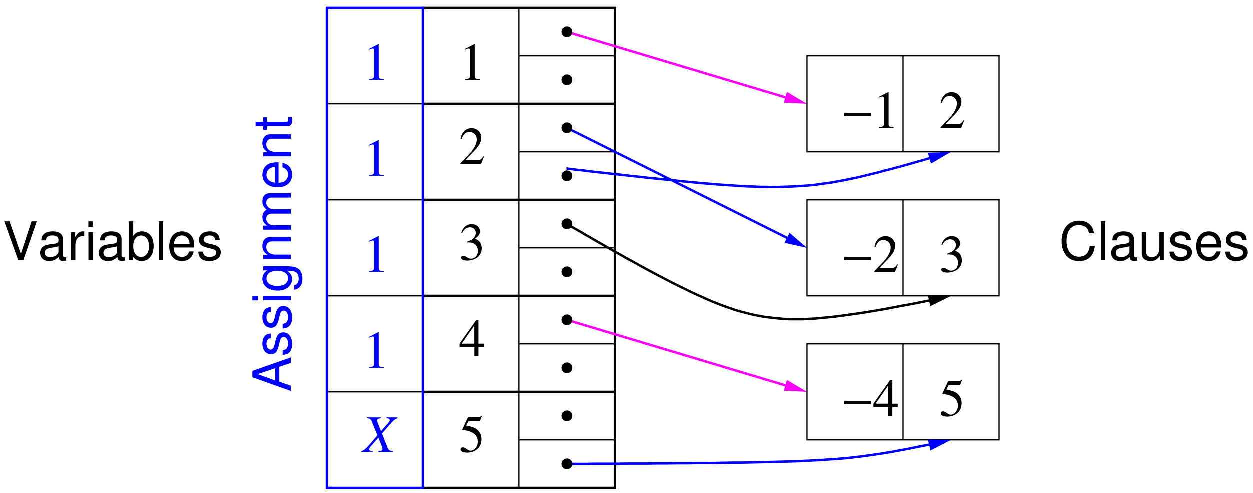
Decide



Example cont.

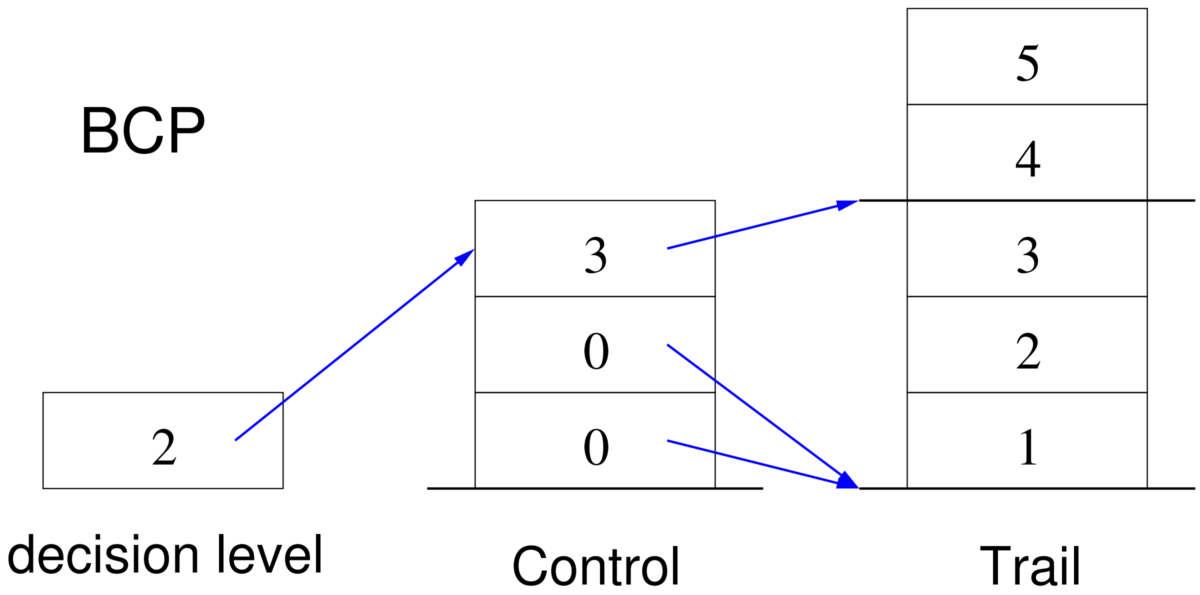
[DavisLogemannLoveland'62]

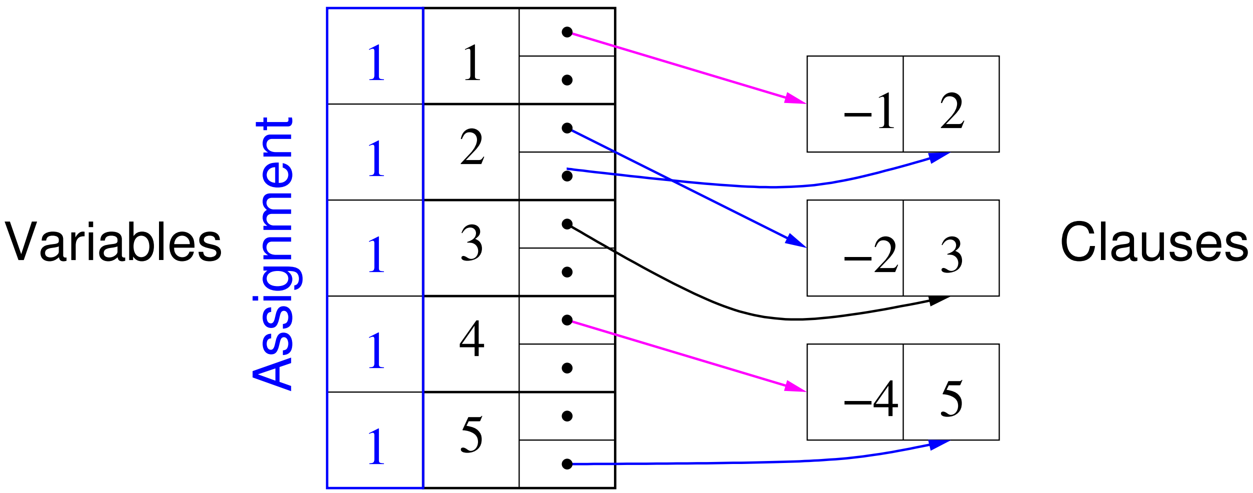




Example cont.

[DavisLogemannLoveland'62]





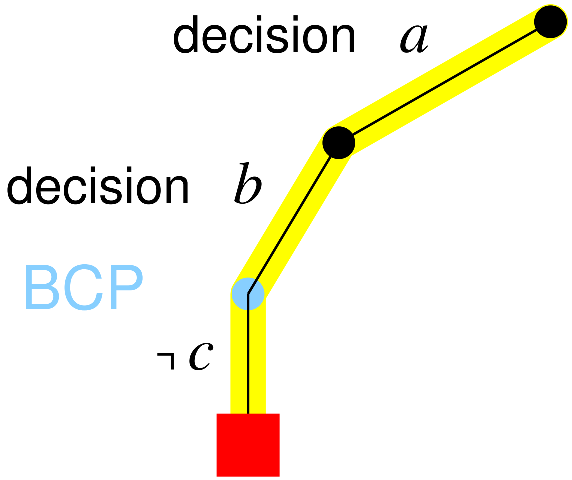
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 1$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

learn $\neg a \vee \neg b$

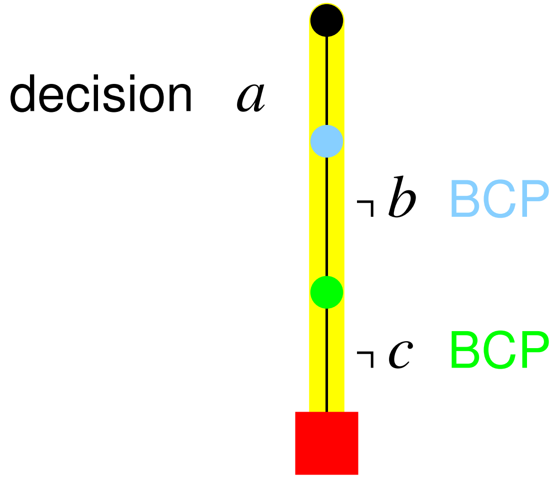
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

learn $\neg a$

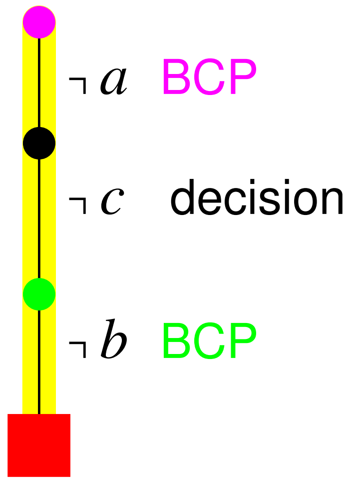
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

learn c

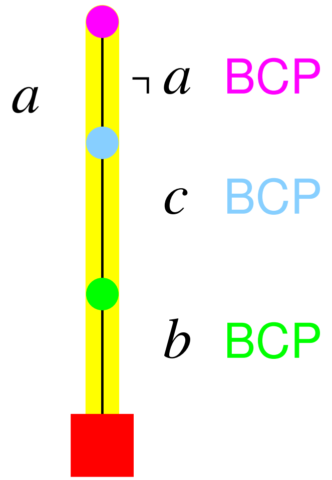
Conflict Driven Clause Learning (CDCL)

Grasp [MarquesSilvaSakallah'96]

$a = 1$

$b = 0$

$c = 0$



clauses

$\neg a \vee \neg b \vee \neg c$

$\neg a \vee \neg b \vee c$

$\neg a \vee b \vee \neg c$

$\neg a \vee b \vee c$

$a \vee \neg b \vee \neg c$

$a \vee \neg b \vee c$

$a \vee b \vee \neg c$

$a \vee b \vee c$

$\neg a \vee \neg b$

$\neg a$

c

learn $\perp$

empty clause

Decision Heuristics

- **static heuristics:**
  - one *linear* order determined before solver is started
  - usually quite fast to compute, since only calculated once
  - and thus can also use more expensive algorithms
- **dynamic heuristics**
  - typically calculated from number of occurences of literals
    (in unsatisfied clauses)
  - could be rather expensive, since it requires traversal of all clauses
    (or more expensive updates in BCP)
  - effective *second order* dynamic heuristics (e.g. VSIDS in Chaff)

Other popular Decision Heuristics

- Dynamic Largest Individual Sum (DLIS)
  - fastest dynamic *first order* heuristic (e.g. GRASP solver)
  - choose literal (variable + phase) which occurs most often (ignore satisfied clauses)
  - requires explicit traversal of CNF (or more expensive BCP)

- look-ahead heuristics (e.g. SATZ or MARCH solver) **failed literals, probing**
  - trial assignments and BCP for all/some unassigned variables (both phases)
  - if BCP leads to conflict, enforce toggled assignment of current trial decision
  - optionally learn binary clauses and perform equivalent literal substitution
  - decision: most balanced w.r.t. prop. assignments / sat. clauses / reduced clauses
  - related to our recent Cube & Conquer paper [HeuleKullmanWieringaBiere'11]

Exponential VSIDS (EVSIDS)

Chaff [MoskewiczMadiganZhaoZhangMalik'01]

- increment score of involved variables by 1
- decay score of all variables every 256'th conflict by halfing the score
- sort priority queue after decay and not at every conflict

MiniSAT uses EVSIDS [EénSörensson'03/'06]

- update score of involved variables — as actually LIS would also do
- dynamically adjust increment: $\delta' = \delta \cdot \frac{1}{f}$ — δ typically increment by $5\% - 11\%$
- use floating point representation of score
- "rescore" to avoid overflow in regular intervals
- EVSIDS linearly related to NVSIDS

Normalized VSIDS: NVSIDS

- VSIDS score can be normalized to the interval [0,1] as follows:
  - pick a decay factor f per conflict: typically $f = 0.9$
  - each variable is punished by this decay factor at every conflict
  - if a variable is involved in conflict, add $1-f$ to score
  - s old score of one fixed variable before conflict, s' new score after conflict

$$s, f \leq 1, \quad \text{then} \quad s' \leq \overbrace{s \cdot f}^{\text{decay in any case}} \underbrace{+1-f}_{\text{increment if involved}} \leq f + 1 - f = 1$$

- recomputing score of all variables at each conflict is costly
  - linear in the number of variables, e.g. millions
  - particularly, because number of involved variabels $<<$ number of variables

Relating EVSIDS and NVSIDS

consider again only one variable with score sequence s_n resp. S_n

$$\delta_k = \begin{cases} 1 & \text{if involved in } k\text{-th conflict} \\ 0 & \text{otherwise} \end{cases}$$

$$i_k = (1-f)\cdot\delta_k$$

$$s_n = \boxed{(\ldots(i_1\cdot f + i_2)\cdot f + i_3)\cdot f \cdots)\cdot f + i_n} = \sum_{k=1}^{n} i_k \cdot f^{n-k} = (1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} \quad \text{(NVSIDS)}$$

$$S_n = \frac{f^{-n}}{(1-f)}\cdot s_n = \frac{f^{-n}}{(1-f)}\cdot(1-f)\cdot\sum_{k=1}^{n}\delta_k\cdot f^{n-k} = \sum_{k=1}^{n}\delta_k\cdot f^{-k} \quad \text{(EVSIDS)}$$

BerkMin's Dynamic Second Order Heuristics

[GoldbergNovikov-DATE'02]

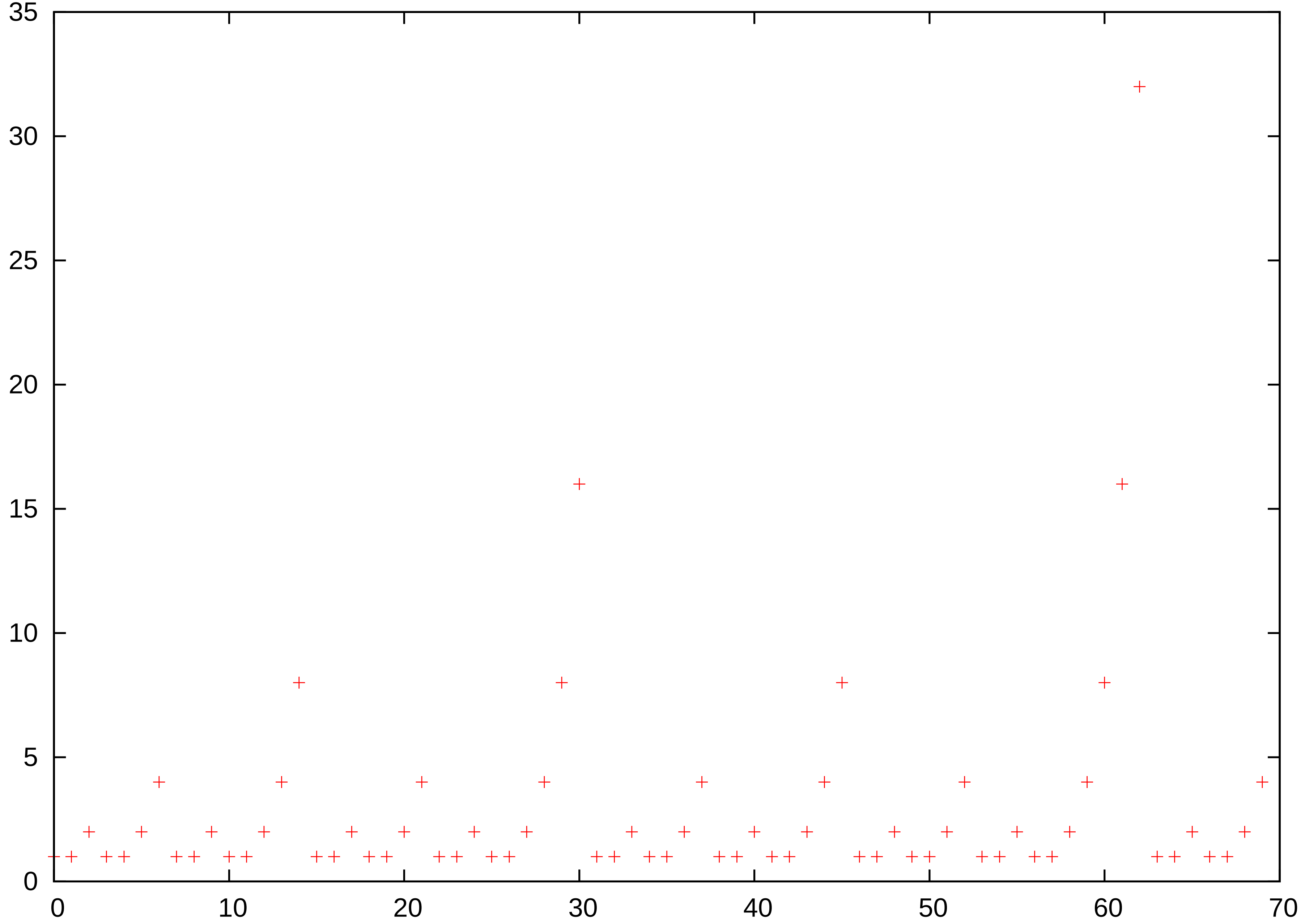
- observation:
  - recently added conflict clauses contain all the good variables of VSIDS
  - the order of those clauses is not used in VSIDS
- basic idea:
  - simply try to satisfy recently learned clauses first
  - use VSIDS to choose the decision variable for one clause
  - if all learned clauses are satisfied use other heuristics
- mixed results as other variants VMTF, CMTF (var/clause move to front)

Restarts

- for satisfiable instances the solver may get stuck in the unsatisfiable part
  - even if the search space contains a large satisfiable part
- often it is a good strategy to abandon the current search and restart
  - restart after the number of decisions reached a *restart limit*
- avoid to run into the same dead end
  - by randomization (either on the decision variable or its phase)
  - and/or just keep all the learned clauses
- for completeness dynamically increase restart limit

Luby's Restart Intervals

70 restarts in 104448 conflicts



Luby Restart Scheduling


```
unsigned
luby (unsigned i)
{
  unsigned k;

  for (k = 1; k < 32; k++)
    if (i == (1 << k) - 1)
      return 1 << (k - 1);

  for (k = 1;; k++)
    if ((1 << (k - 1)) <= i && i < (1 << k) - 1)
      return luby (i - (1 << (k-1)) + 1);
}

limit = 512 * luby (++restarts);
...  // run SAT core loop for 'limit' conflicts
```

Reluctant Doubling Sequence

[Knuth'12]

$$
\begin{aligned}
(u_1, v_1) &:= (1, 1) \\
(u_{n+1}, v_{n+1}) &:= (u_n \,\&\, {-u_n} = v_n \;?\; (u_n + 1, 1) : (u_n, 2v_n))
\end{aligned}
$$

$$(1,1),\ (2,1),\ (2,2),\ (3,1),\ (4,1),\ (4,2),\ (4,4),\ (5,1),\ \ldots$$

Phase Saving and Rapid Restarts

- phase assignment / direction heuristics:
  - assign decision variable to 0 or 1?
  - only thing that matters in *satisfiable* instances
- "phase saving" as in RSat:
  - pick phase of last assignment (if not forced to, do not toggle assignment)
  - initially use statically computed phase (typically LIS)
  - so can be seen to maintain a *global full assignment*
- rapid restarts: varying restart interval with bursts of restarts
  - not ony theoretically avoids local minima
  - empirically works nice together with phase saving

Reusing the Trail

[Van Der Tak, Heule, Ramos POS'11]

- in general *restarting does not much change much*: since phases and scores saved
- assignment after restart can only differ if
  - before restarting
  - there is a decision literal d assigned on the trail
  - with smaller score than the next decision n on the priority queue
- in this situation backtrack **only** to decision level of d
  - simple to compute, particularly if decisions are saved separately
  - allows to skip many redundant backtracks
  - allows much higer restart frequency,
    e.g. base interval 10 for reluctant doubling sequence (Luby)

Reducing Learned Clauses

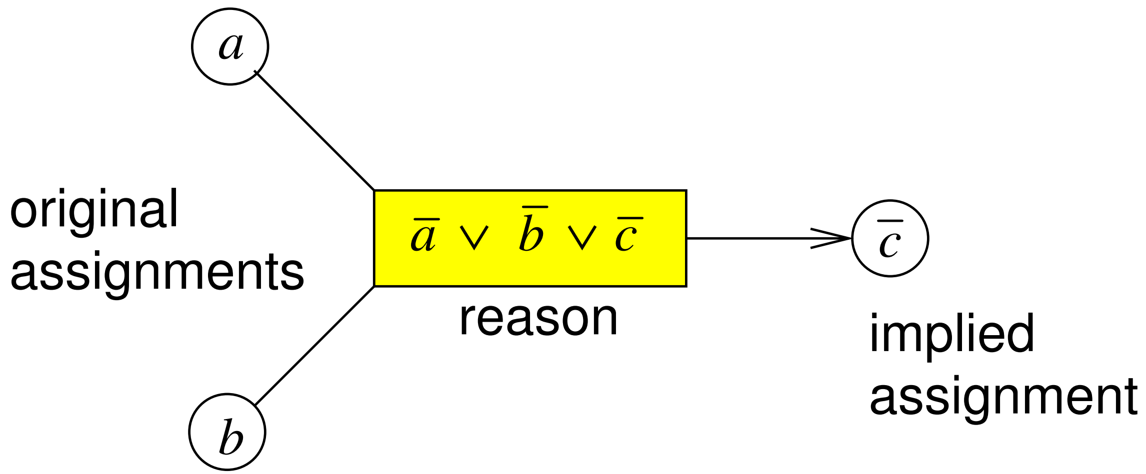
- keeping all learned clauses slows down BCP — kind of quadratically
  - so SATO and RelSAT just kept only “short” clauses

- better periodically delete “useless” learned clauses
  - keep a certain number of learned clauses — “search cache”
  - if this number is reached MiniSAT reduces (deletes) half of the clauses
  - keep *most active*, then *shortest*, then *youngest* (FIFO) clauses
  - after reduction maximum number kept learned clauses is increased geometrically

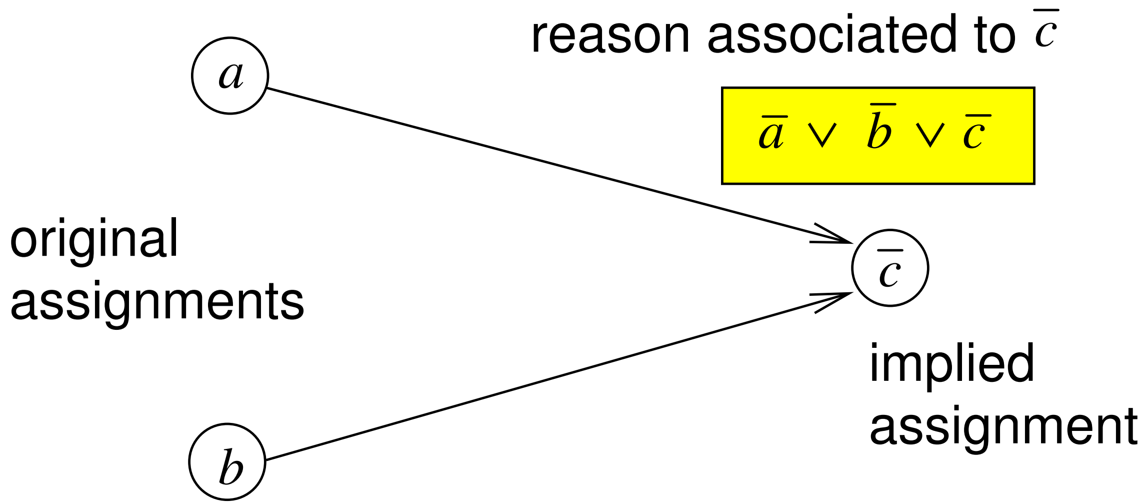
- LBD (Glue) based (apriori!) prediction for usefullness — [AudemardSimon’09]
  - LBD (Glue) = number of decision-levels in the learned clause
  - allows arithmetic increase of number of kept learned clauses

- freeze high PSM (dist. to phase assign.) clauses — [AudemardLagniezMazureSais’11]

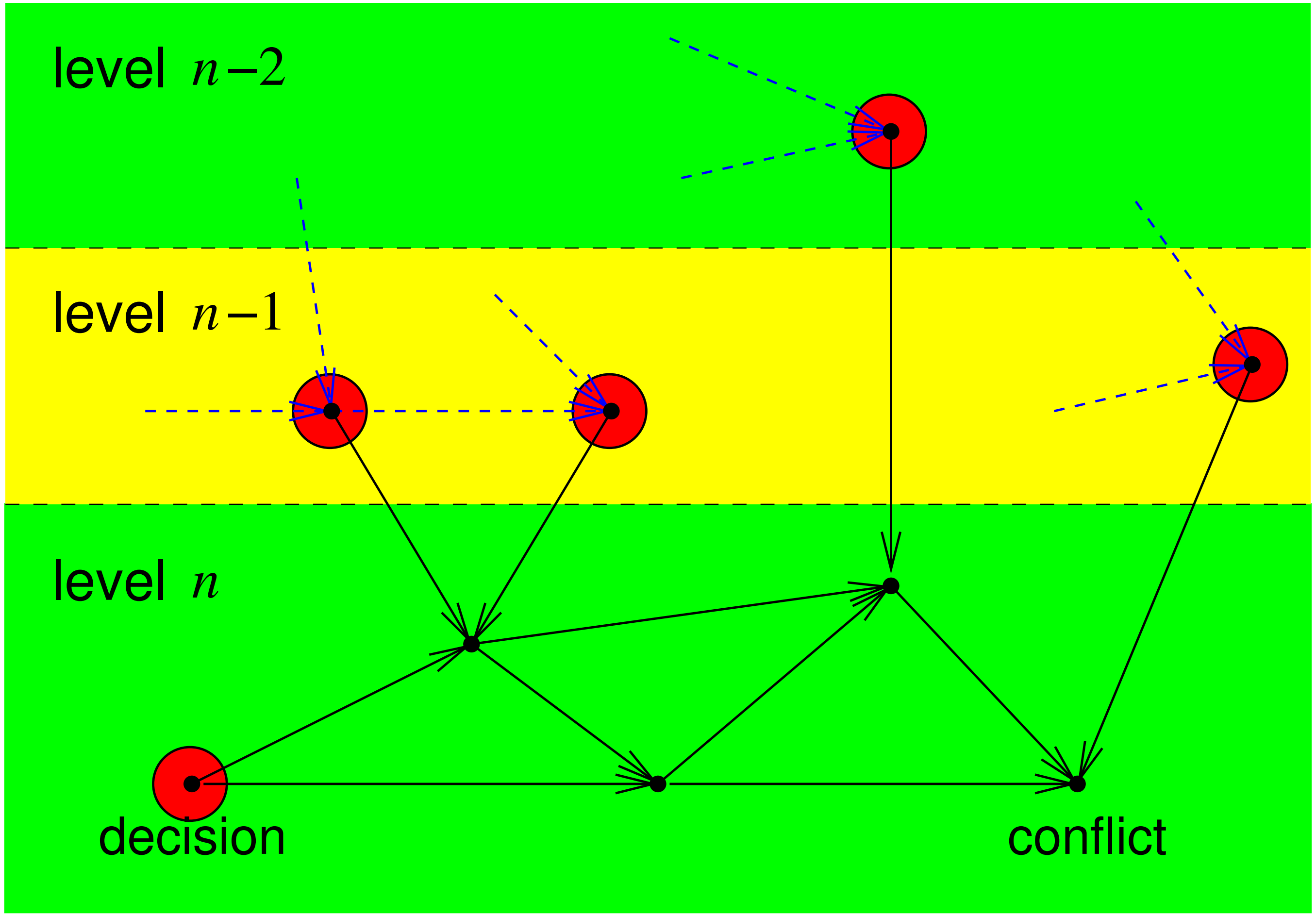
General Implication Graph as Hyper-Graph



Implication Graph Standard Notation



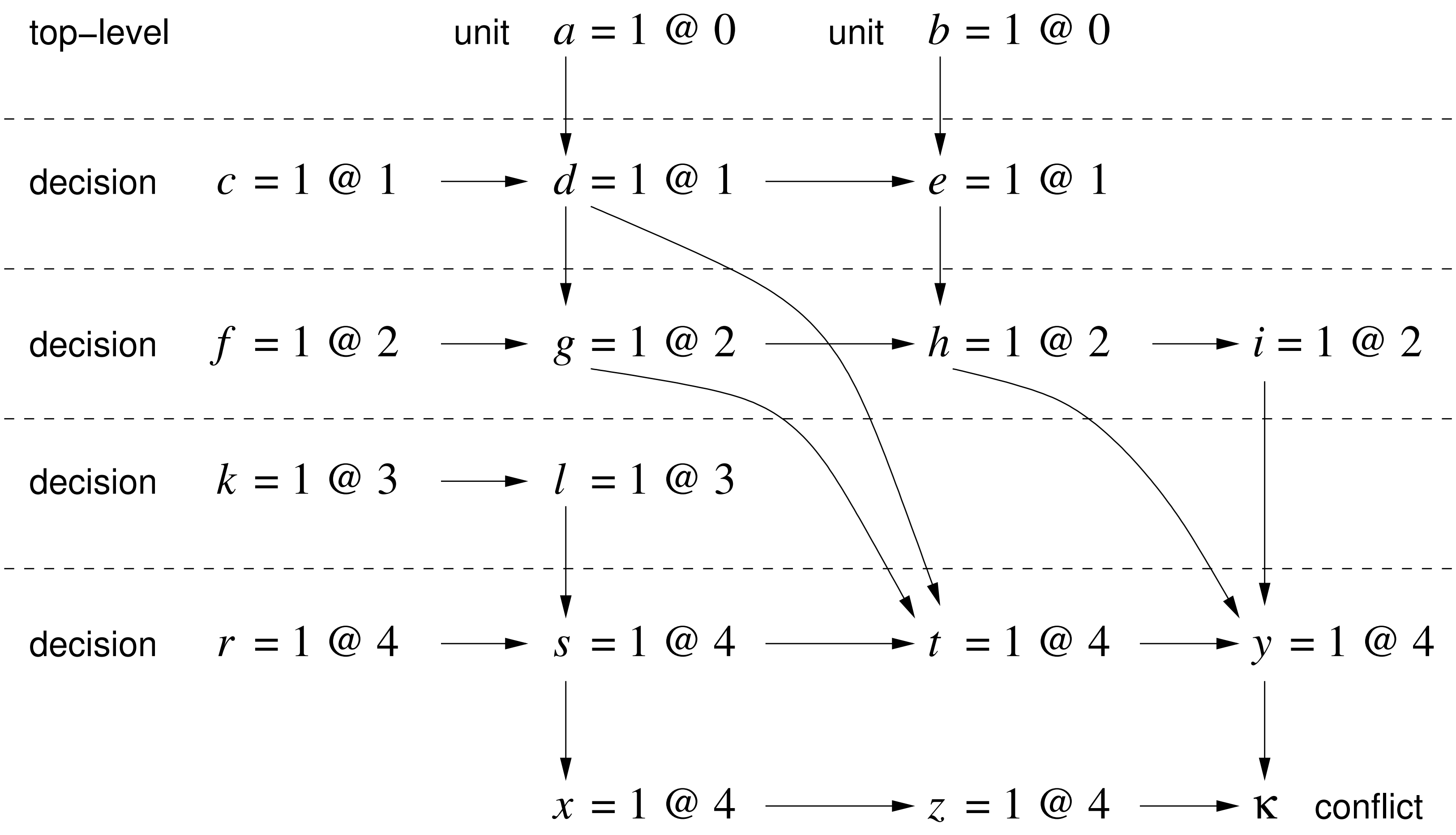
Conflict Clauses as Cuts in the Implication Graph



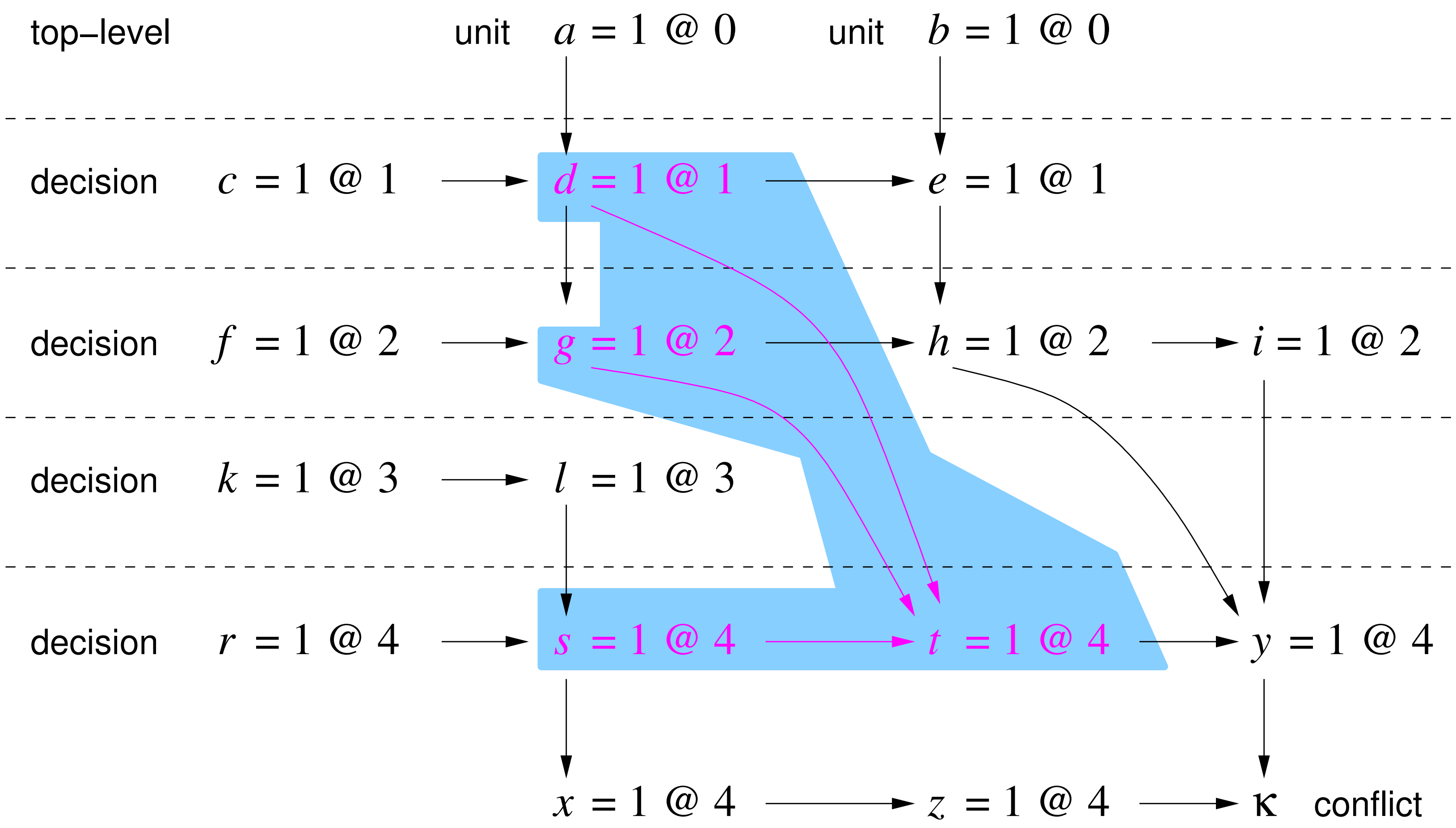
a simple cut always exists: set of roots (decisions) contributing to the conflict

Implication Graph

CDCL / Grasp [MarquesSilvaSakallah'96]



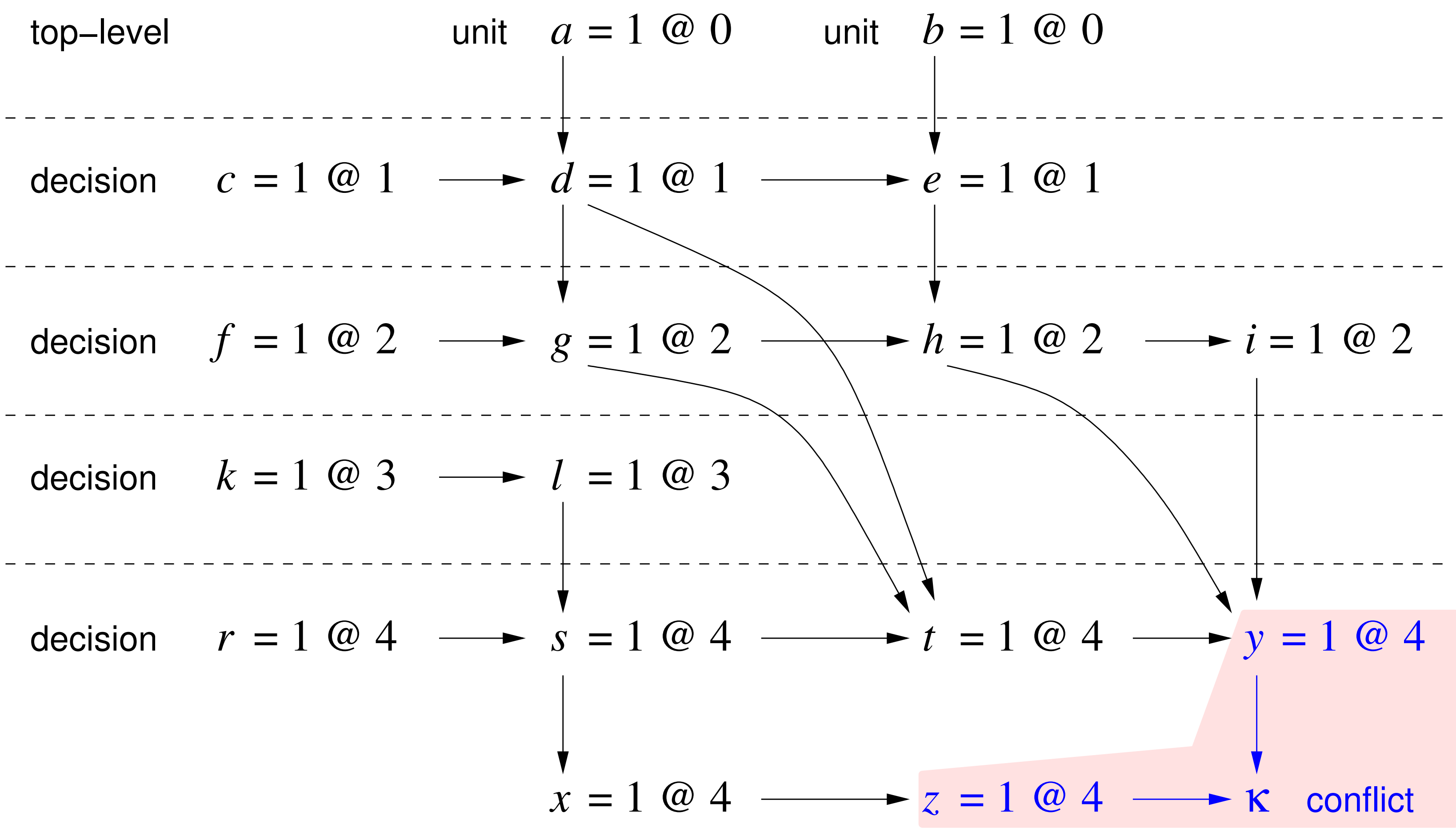
Antecedents / Reasons



$$d \wedge g \wedge s \rightarrow t \qquad \equiv \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee t)$$

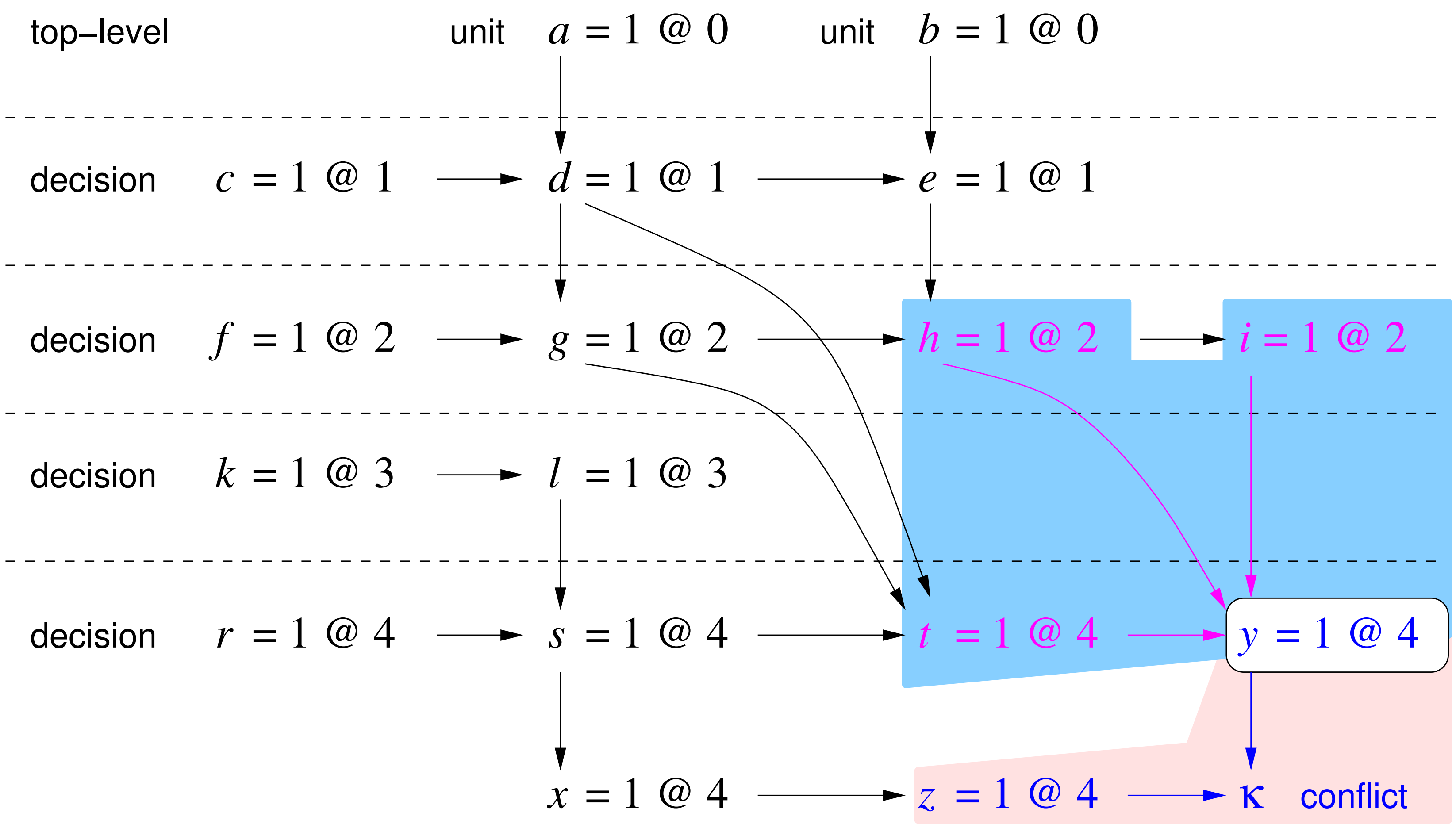
Conflicting Clauses

CDCL / Grasp [MarquesSilvaSakallah'96]



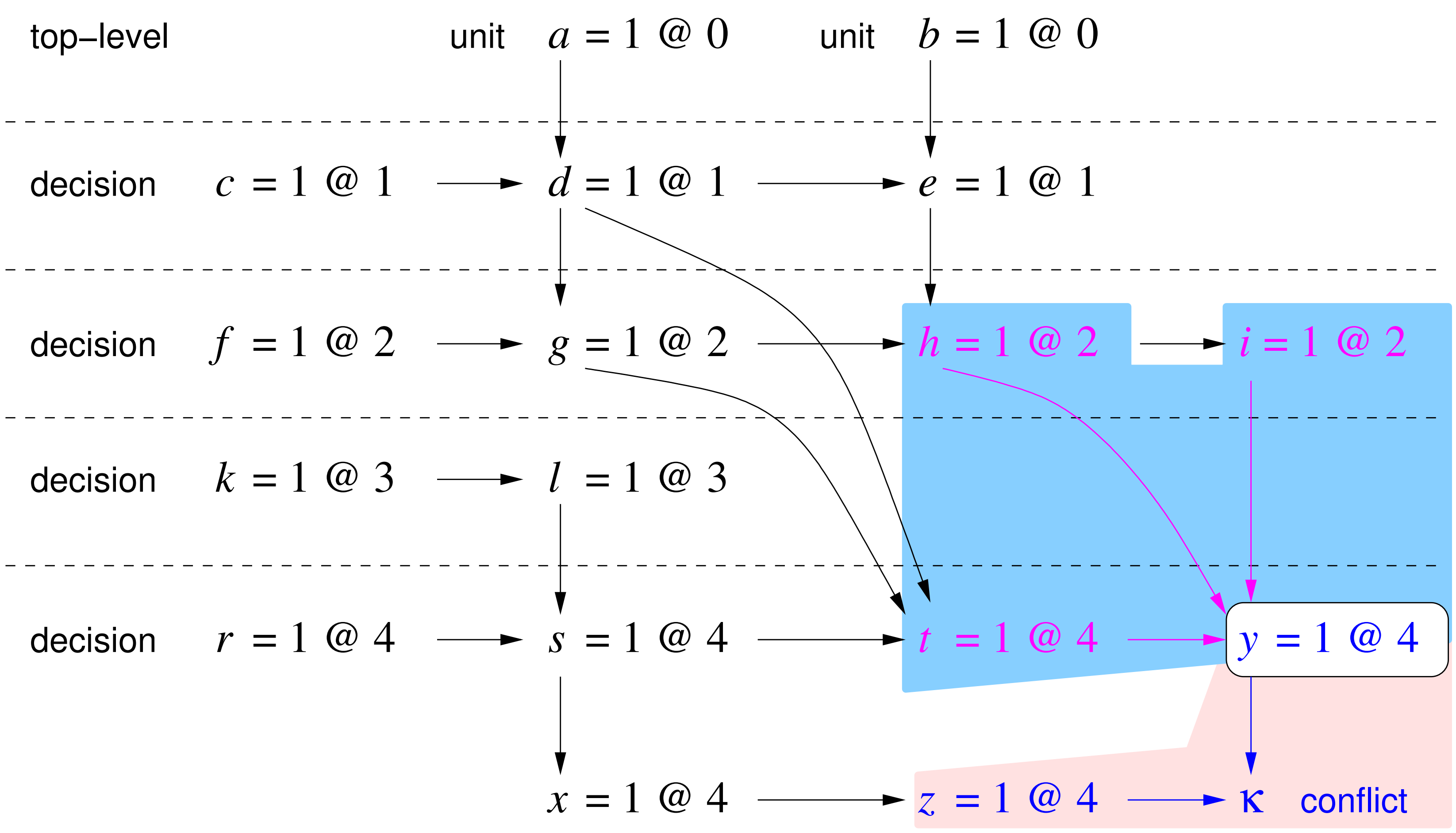
$$\neg(y \wedge z) \qquad \equiv \qquad (\bar{y} \vee \bar{z})$$

Resolving Antecedents 1st Time



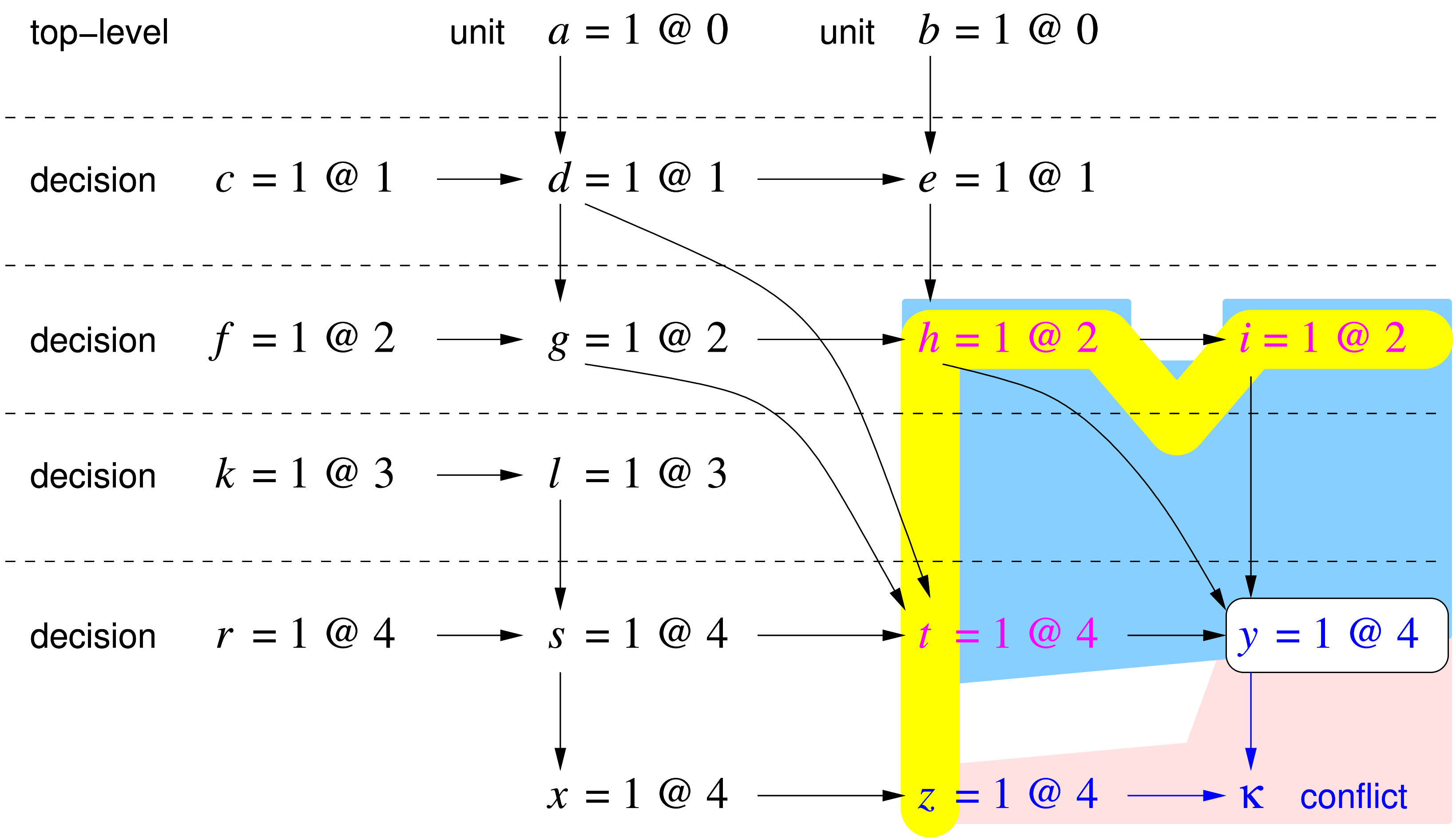
$(\bar{h} \vee \bar{i} \vee \bar{t} \vee y)$ $(\bar{y} \vee \bar{z})$

Resolving Antecedents 1st Time



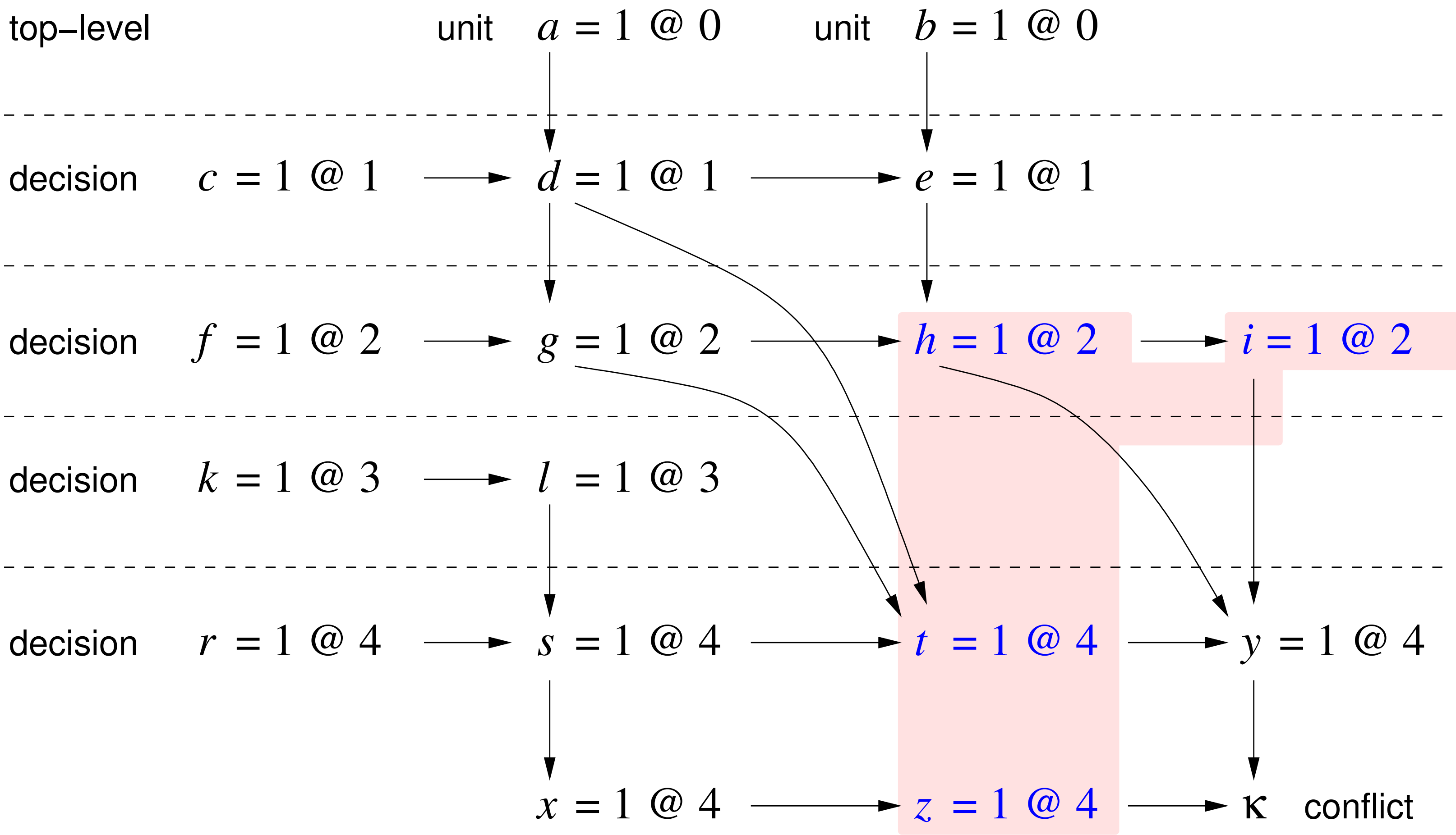
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{\boxed{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}}$$

Resolvents = Cuts = Potential Learned Clauses



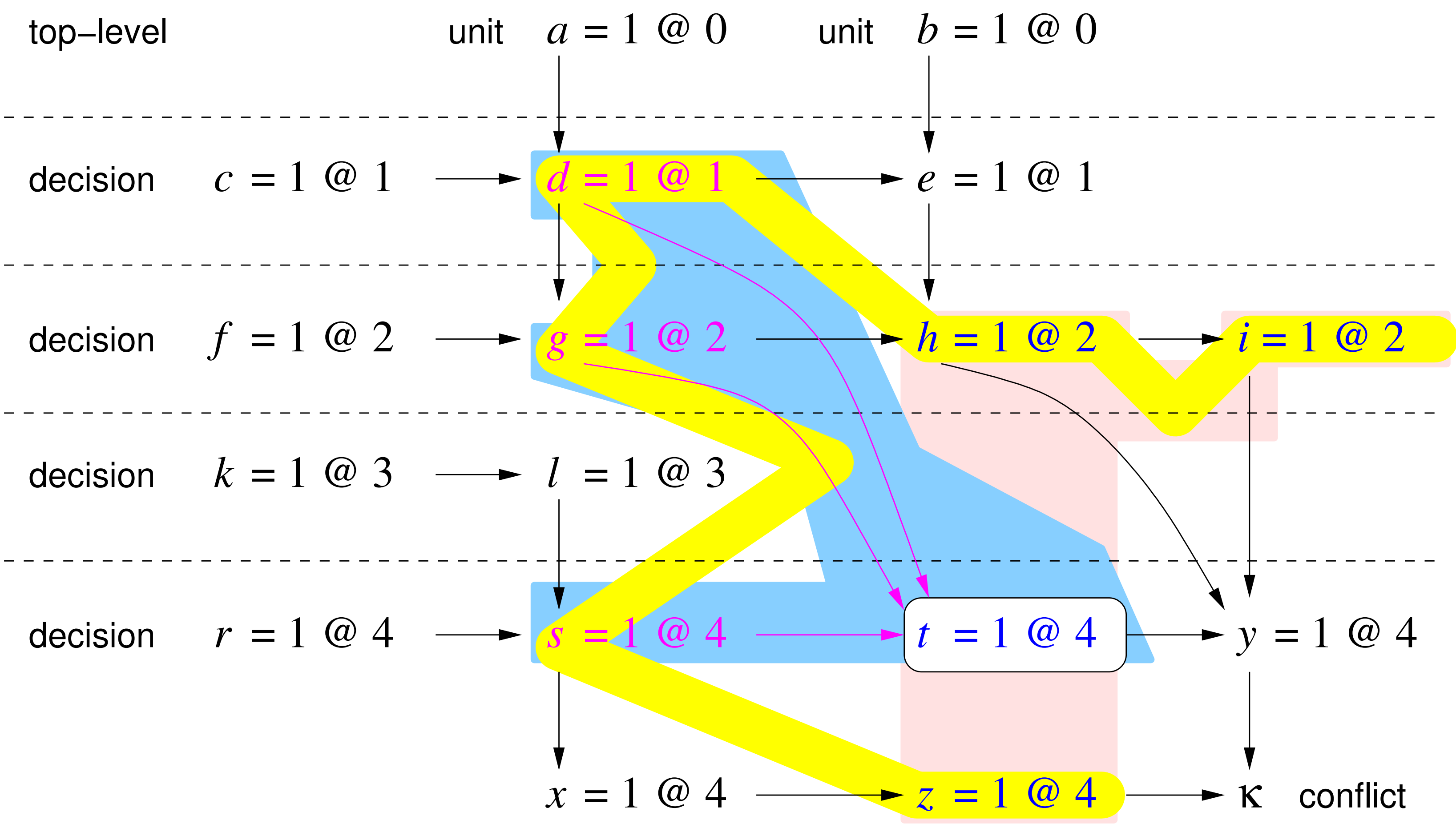
$$\frac{(\bar{h} \vee \bar{i} \vee \bar{t} \vee y) \qquad (\bar{y} \vee \bar{z})}{\boxed{(\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}}$$

Potential Learned Clause After 1 Resolution



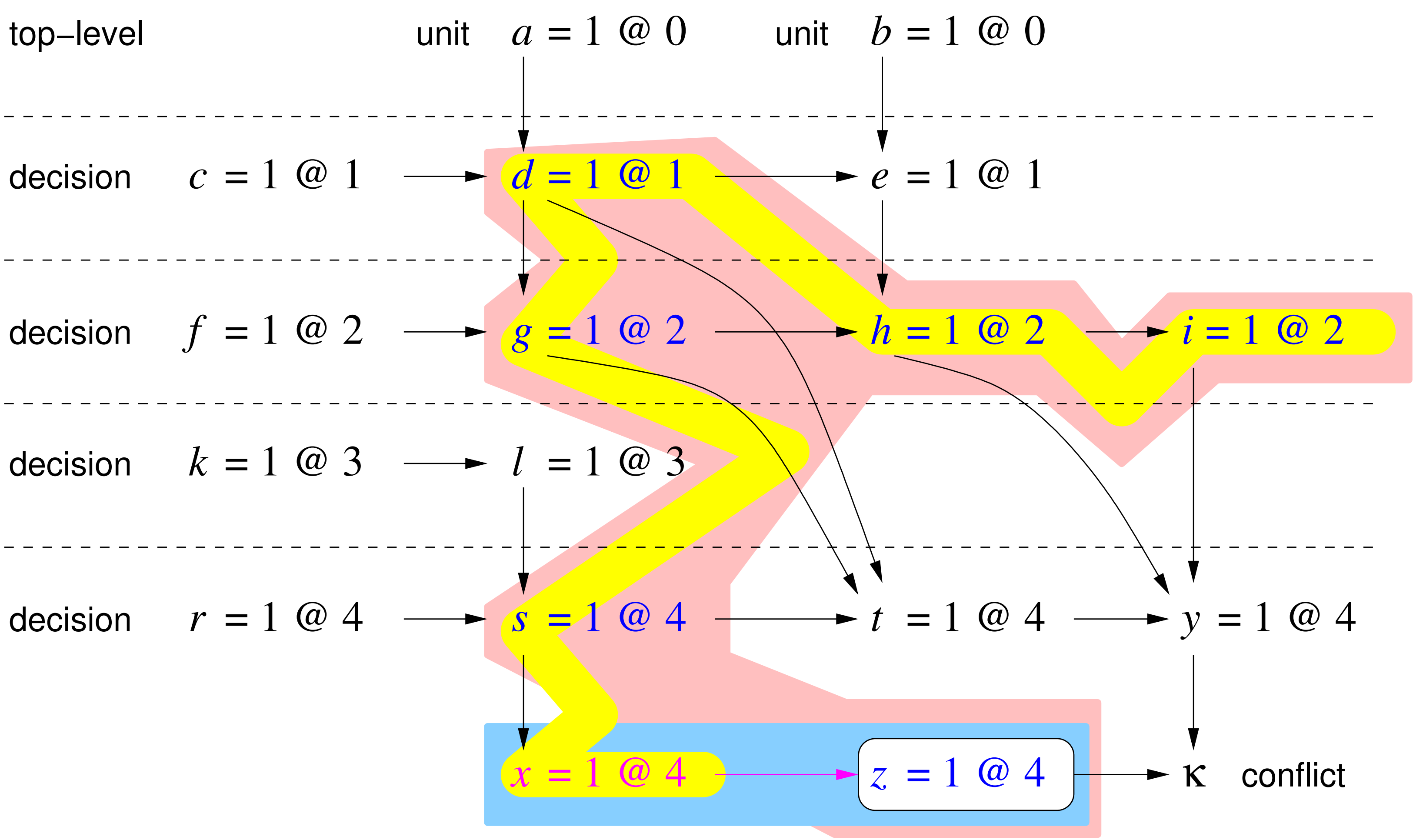
$$(\overline{h} \vee \overline{i} \vee \overline{t} \vee \overline{z})$$

Resolving Antecedents 2nd Time



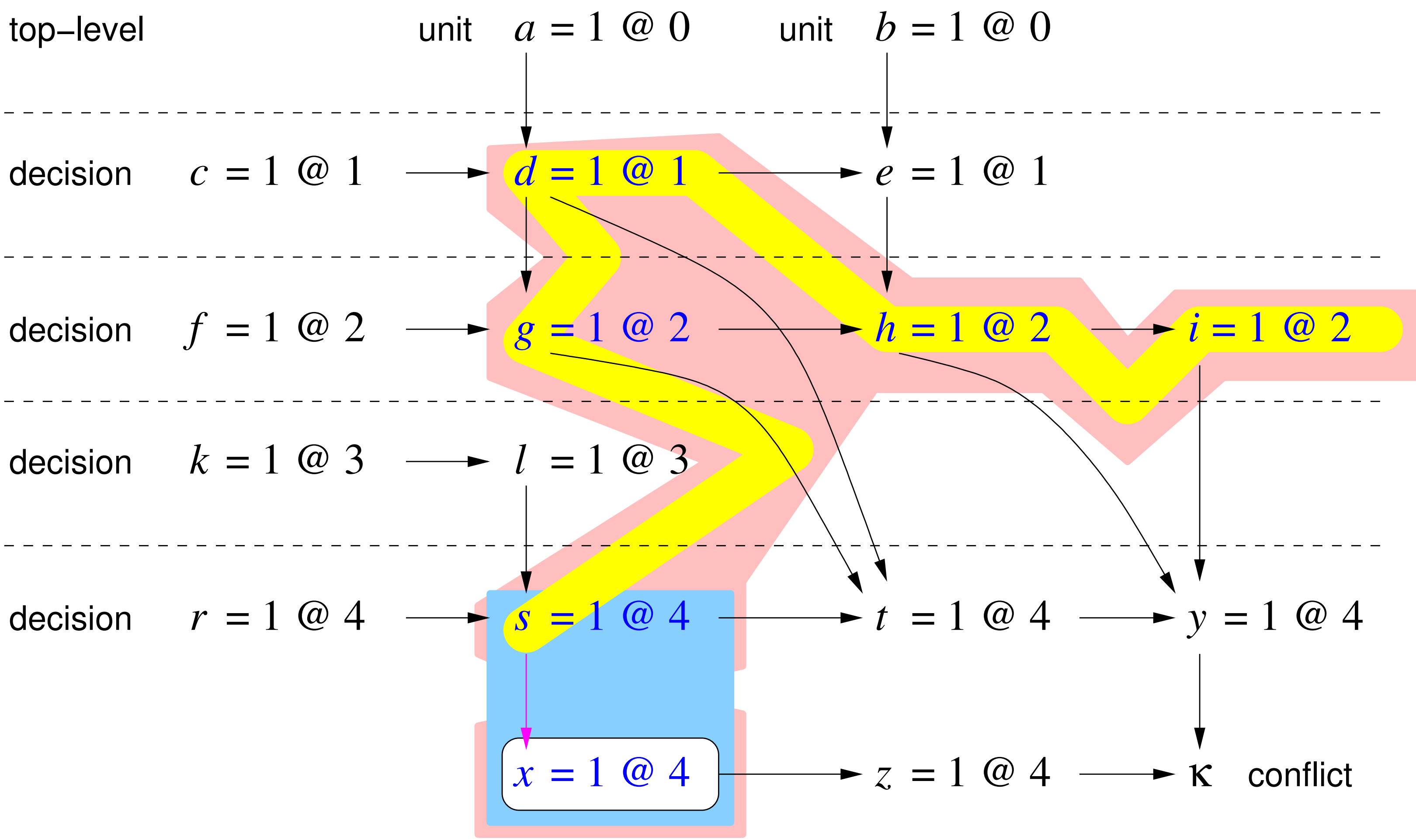
$$\frac{(\bar{d} \vee \bar{g} \vee \bar{s} \vee t) \qquad (\bar{h} \vee \bar{i} \vee \bar{t} \vee \bar{z})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i} \vee \bar{z})}$$

Resolving Antecedents 3rd Time



$$\frac{(\overline{x} \vee z) \qquad (\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i} \vee \overline{z})}{\boxed{(\overline{x} \vee \overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})}}$$

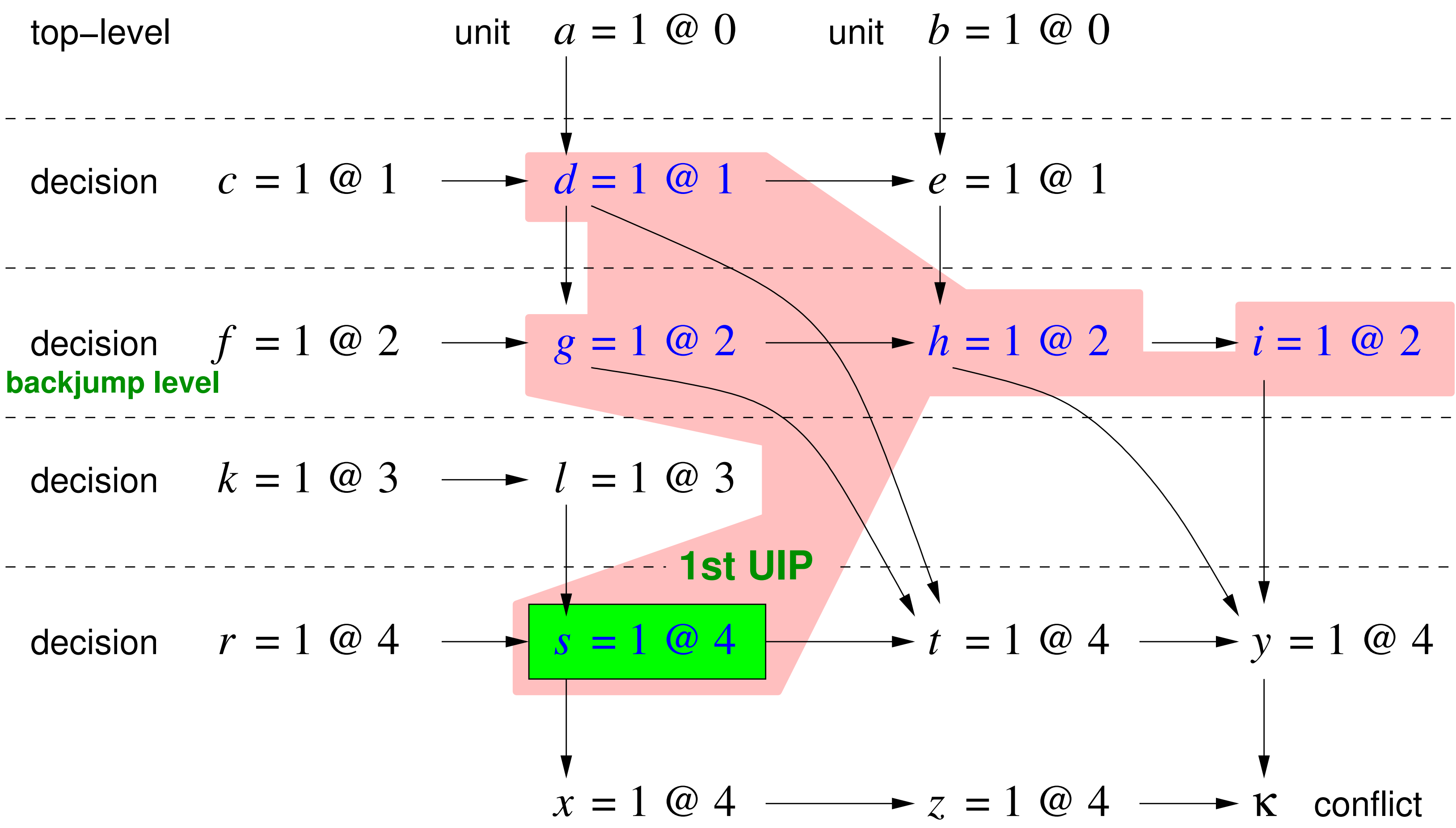
Resolving Antecedents 4^{th} Time



$$\frac{(\bar{s} \vee x) \qquad (\bar{x} \vee \bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}$$

self subsuming resolution

1^{st} UIP Clause after 4 Resolutions



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

UIP = *unique implication point* dominates conflict on the last level

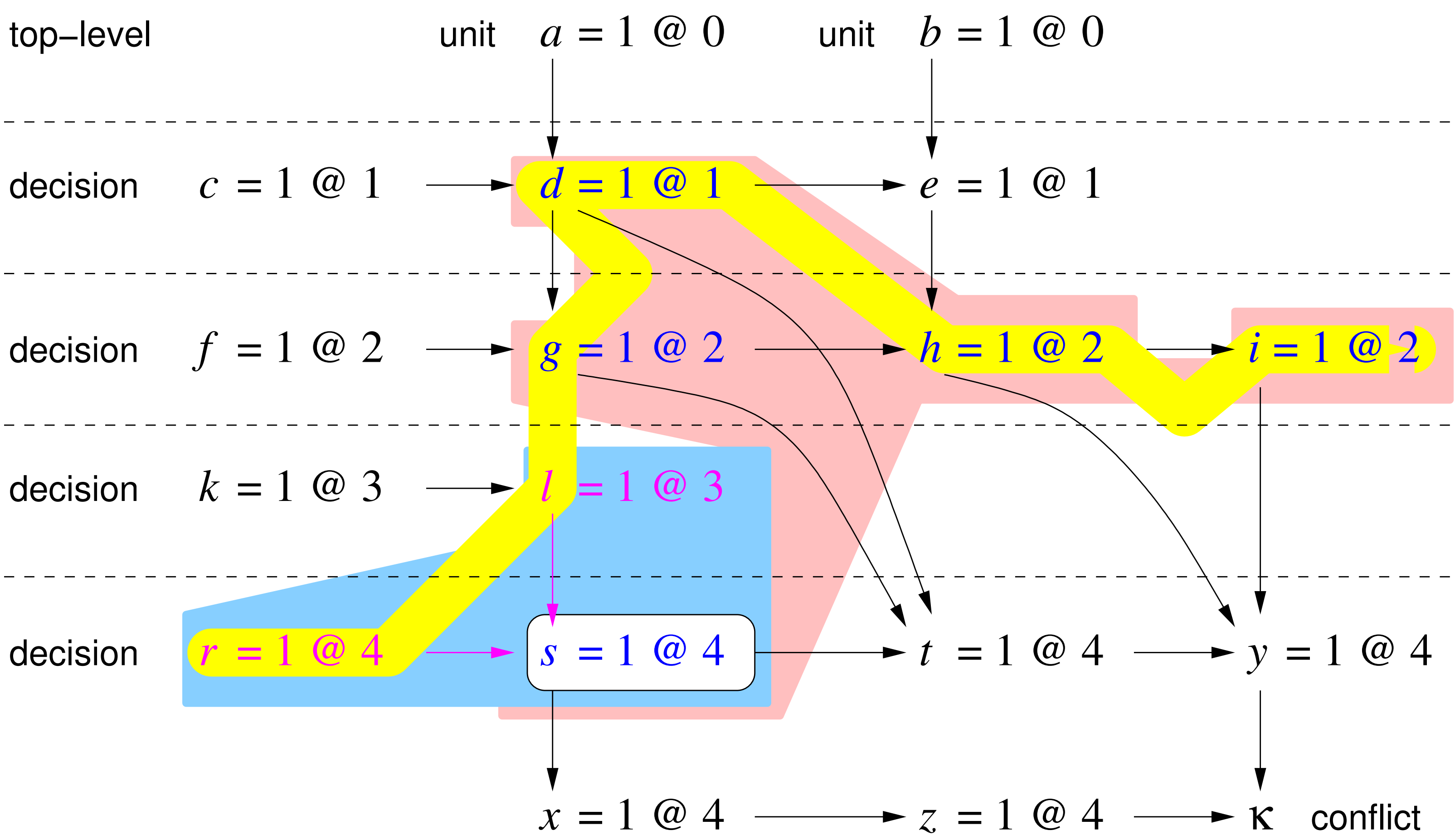
Simple Algorithm to Find First UIP Clause

- can be found by *graph traversal* in the reverse order of made assignments
  - *trail* respects this order
  - mark literals in conflict
  - traverse reasons of marked variables on trail in reverse order
- count *number unresolved variables* on current decision level
  - decrease counter if new reason / antecedent clause resolved
  - if counter=1 (only one unresolved marked variable left) then this node is a UIP
  - note, decision of current decision level is a UIP and thus a *sentinel*

Modern CDCL Loop

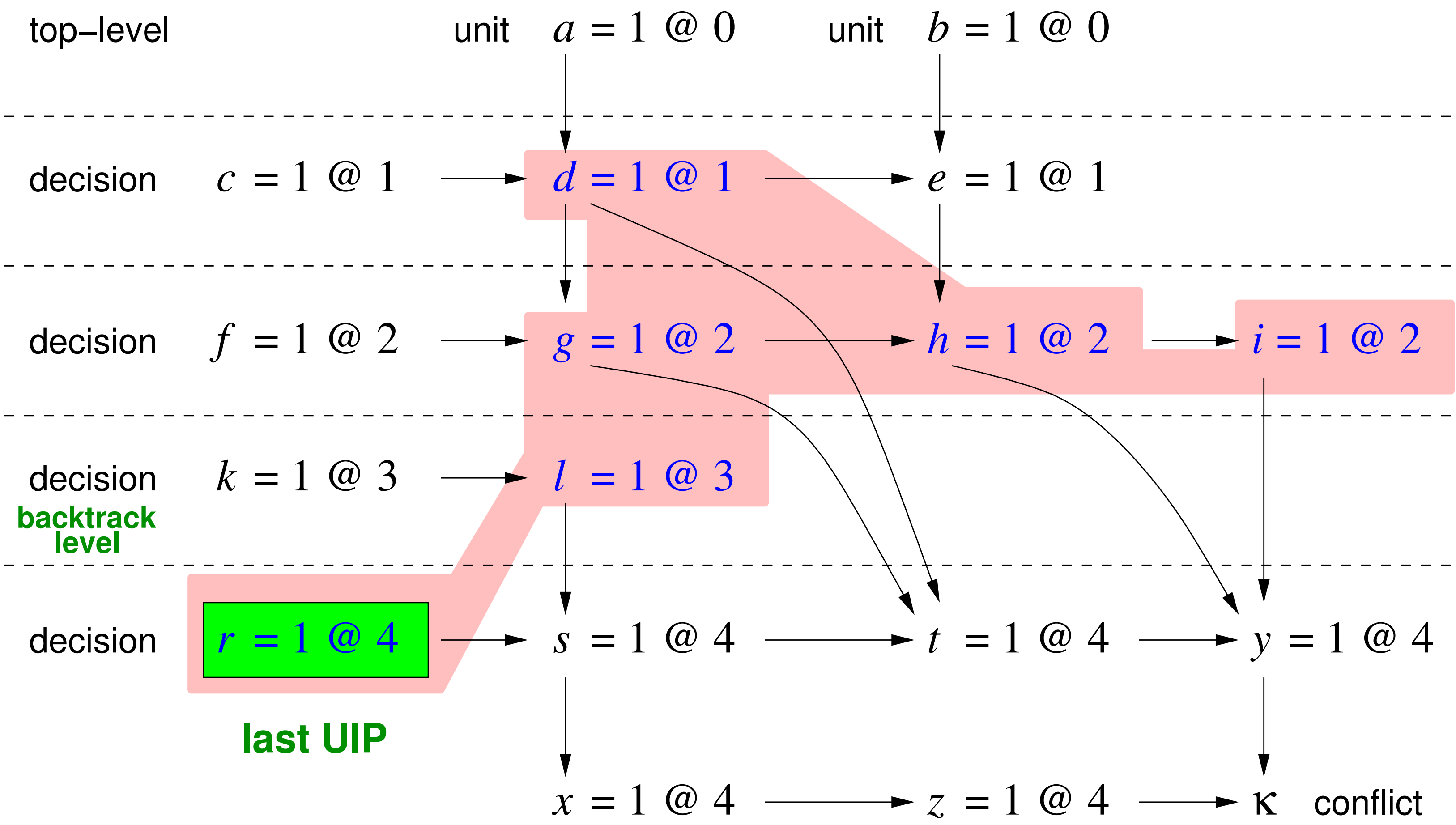

```
Status Solver::search (long limit) {
  long conflicts = 0; Clause * conflict; Status res = UNKNOWN;
  while (!res)
    if (empty) res = UNSATISFIABLE;
    else if ((conflict = bcp ())) analyze (conflict), conflicts++;
    else if (conflicts >= limit) break;
    else if (reducing ()) reduce ();
    else if (restarting ()) restart ();
    else if (!decide ()) res = SATISFIABLE;
  return res;
}

Status Solver::solve () {
  long conflicts = 0, steps = 1e6;
  Status res;
  for (;;)
    if ((res = search (conflicts))) break;
    else if ((res = simplify (steps))) break;
    else conflicts += 1e4, steps += 1e6;
  return res;
}
```

Resolving Antecedents 5th Time



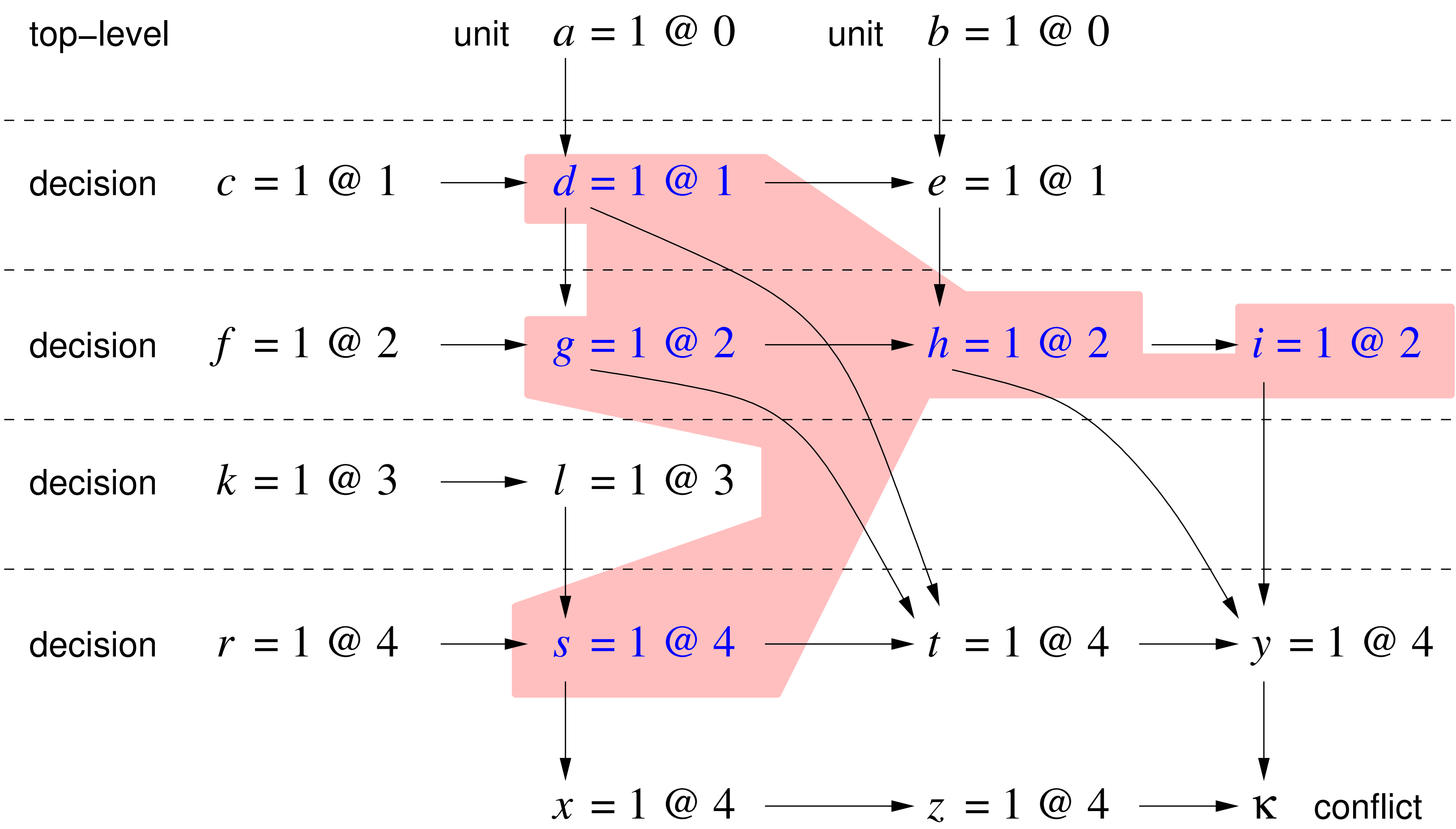
$$\frac{(\bar{l} \vee \bar{r} \vee s) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{l} \vee \bar{r} \vee \bar{d} \vee \bar{g} \vee \bar{h} \vee \bar{i})}$$

Decision Learned Clause



$$(\overline{d} \vee \overline{g} \vee \overline{l} \vee \overline{r} \vee \overline{h} \vee \overline{i})$$

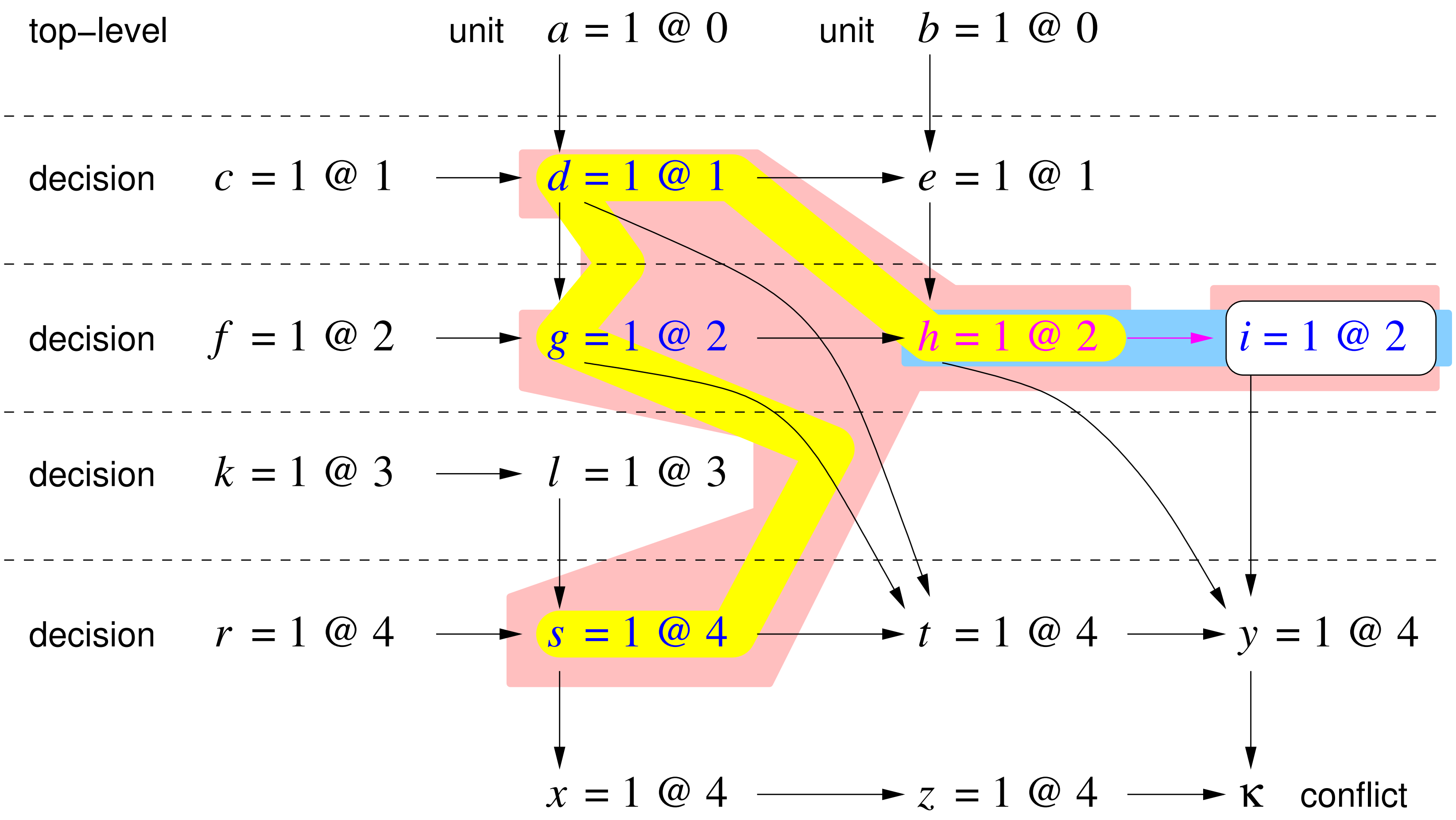
1st UIP Clause after 4 Resolutions



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h} \vee \overline{i})$$

Locally Minimizing 1st UIP Clause

Sörensson'06, BiereSörensson'09

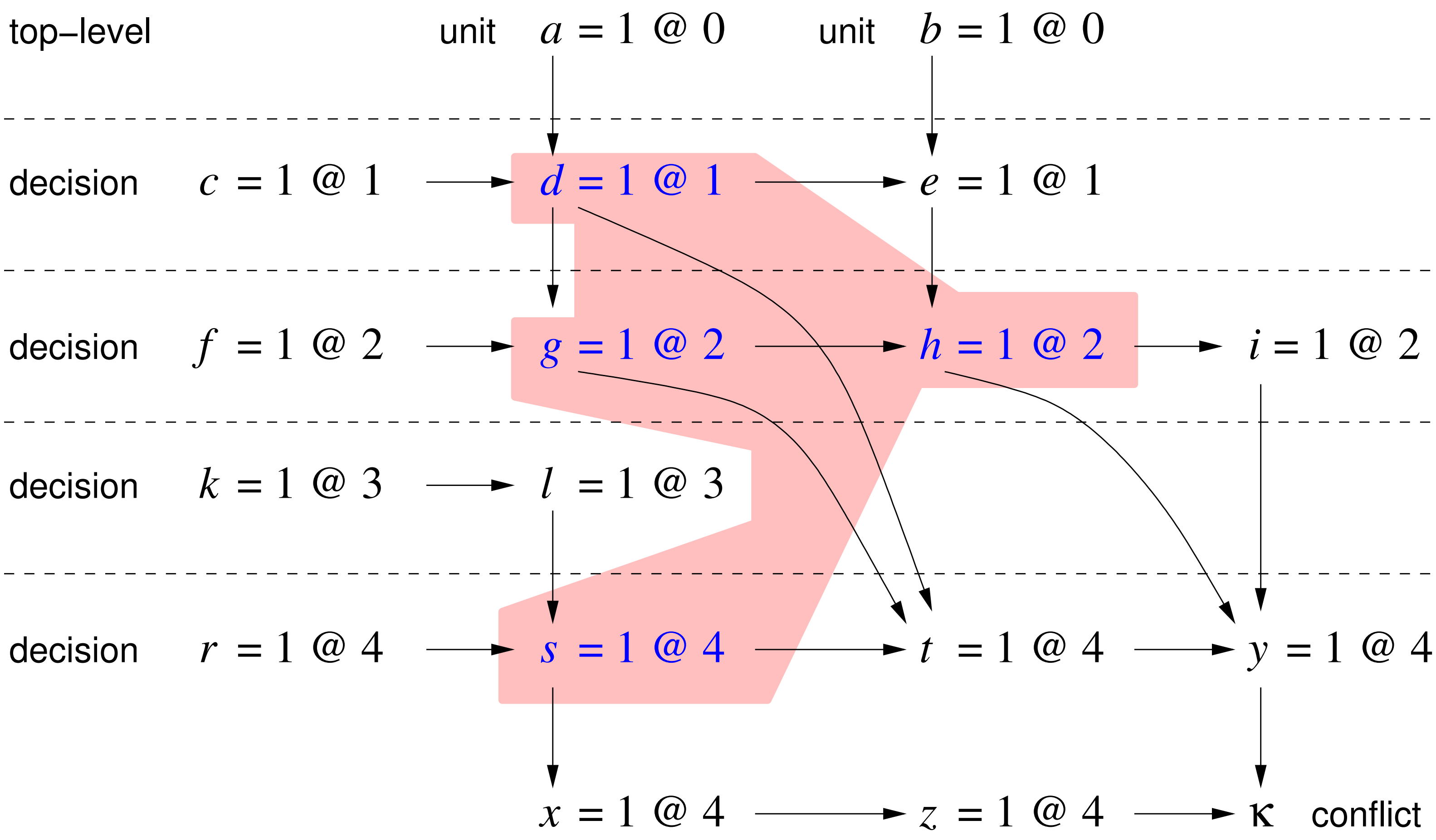


$$\frac{(\bar{h} \vee i) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h} \vee \bar{i})}{(\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h})}$$

self subsuming resolution

Locally Minimized Learned Clause

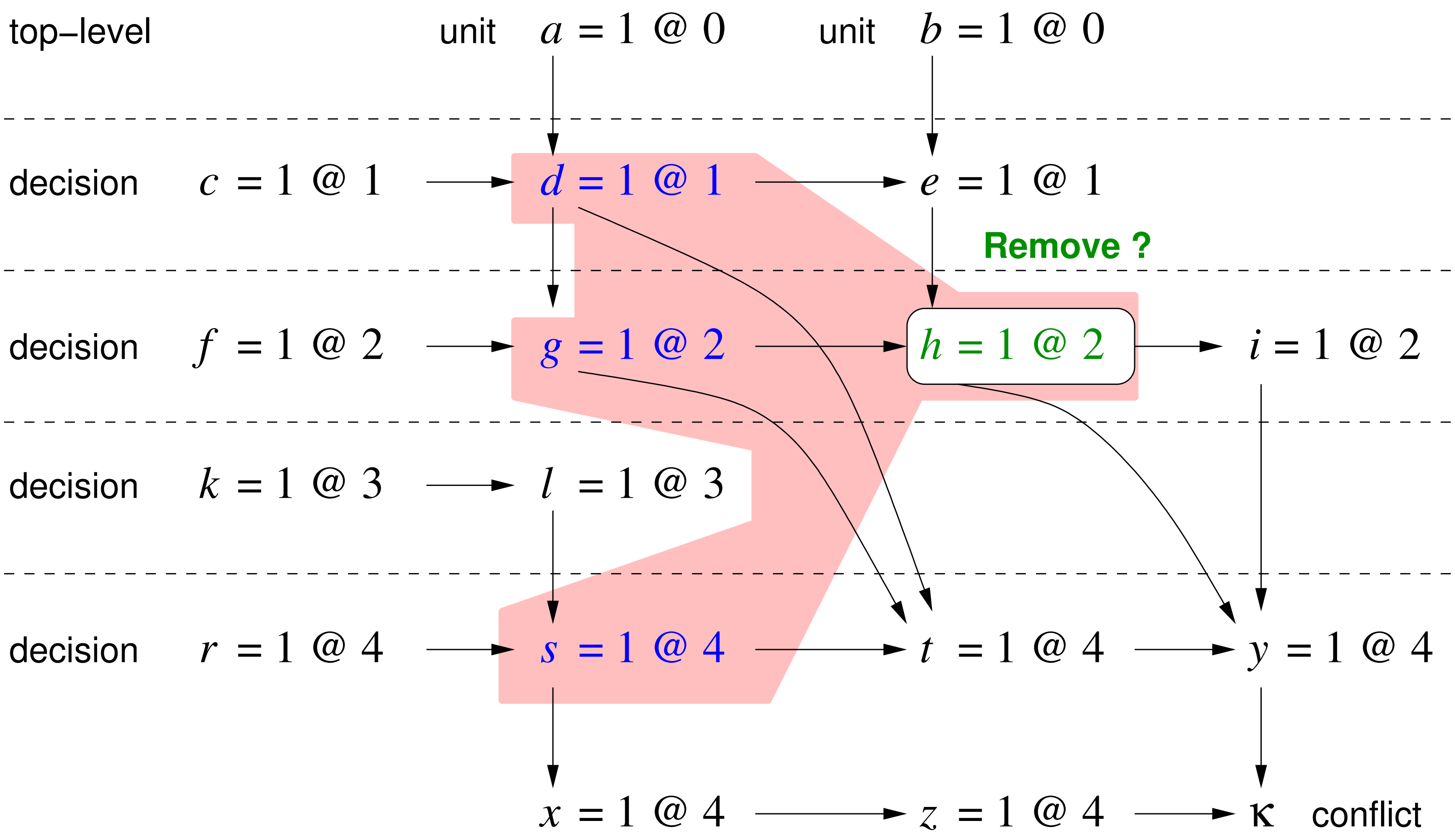
Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \overline{h})$$

Minimizing Locally Minimized Learned Clause Further?

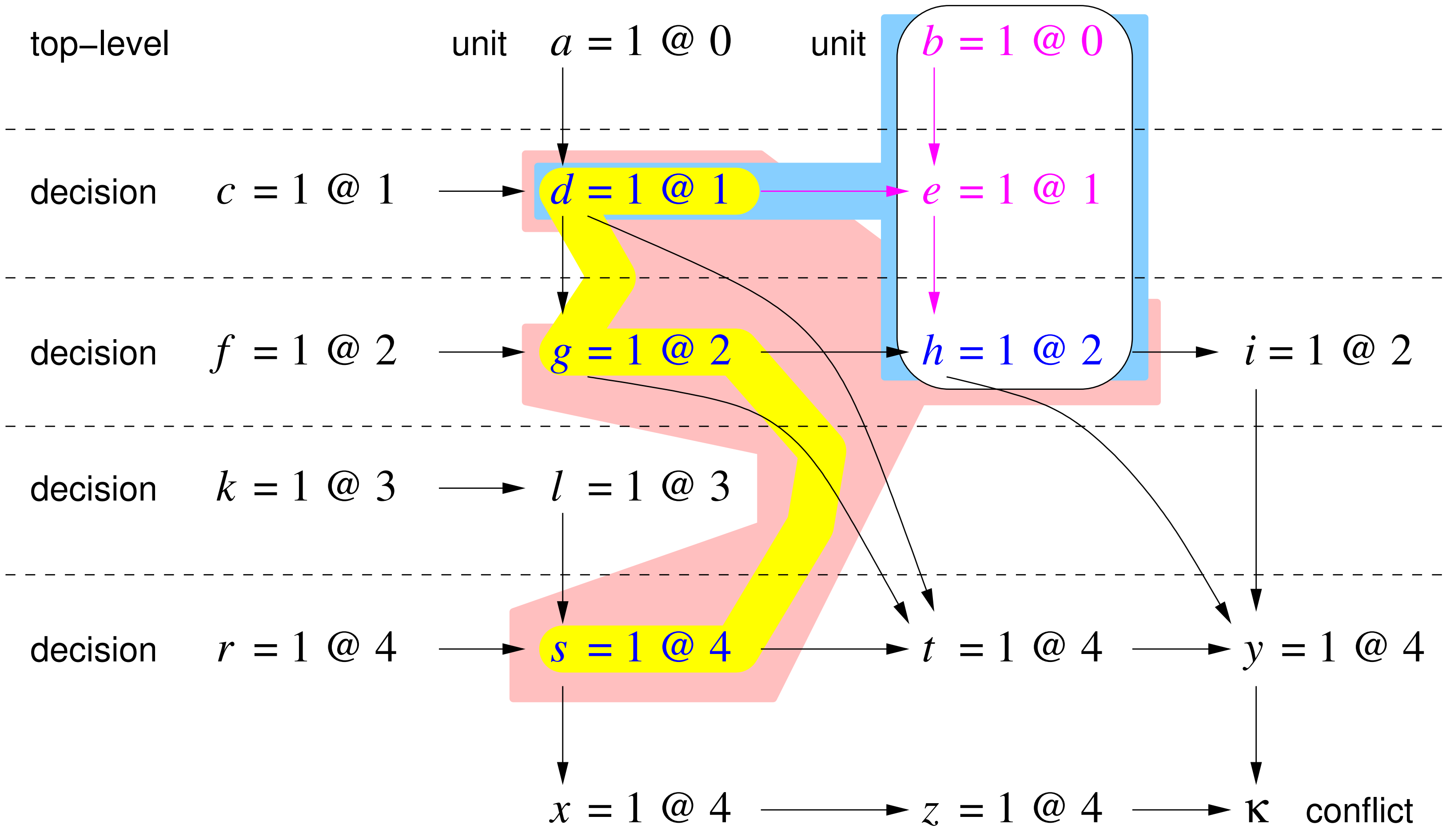
Sörensson'06, BiereSörensson'09



$$(\overline{d} \vee \overline{g} \vee \overline{s} \vee \cancel{\overline{h}})$$

Recursively Minimizing Learned Clause

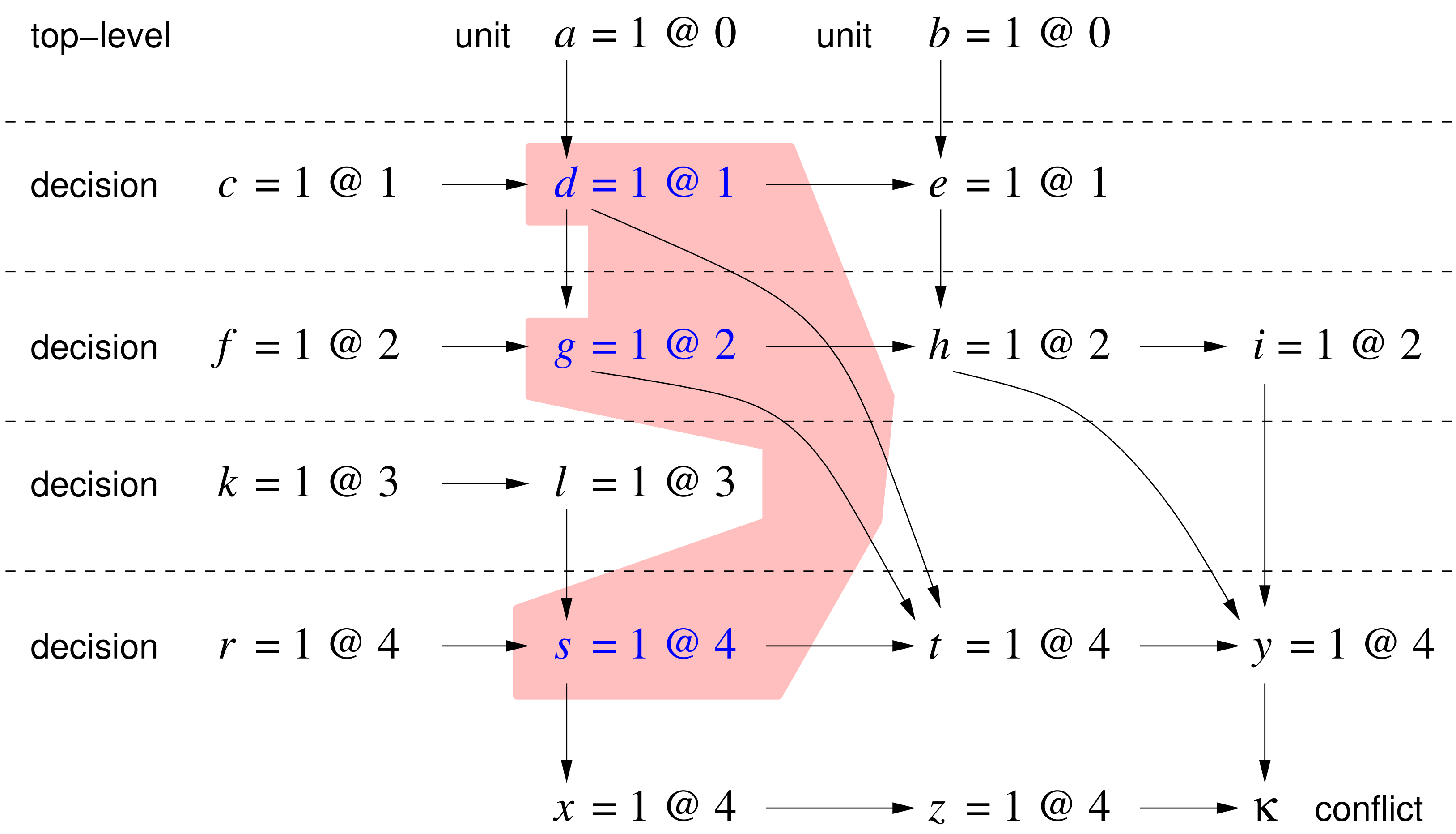
Sörensson'06, BiereSörensson'09



$$\dfrac{(b) \qquad \dfrac{(\bar{d} \vee \bar{b} \vee e) \qquad \dfrac{(\bar{e} \vee \bar{g} \vee h) \qquad (\bar{d} \vee \bar{g} \vee \bar{s} \vee \bar{h})}{(\bar{e} \vee \bar{d} \vee \bar{g} \vee \bar{s})}}{(\bar{b} \vee \bar{d} \vee \bar{g} \vee \bar{s})}}{\boxed{(\bar{d} \vee \bar{g} \vee \bar{s})}}$$

Recursively Minimized Learned Clause

Sörensson'06, BiereSörensson'09



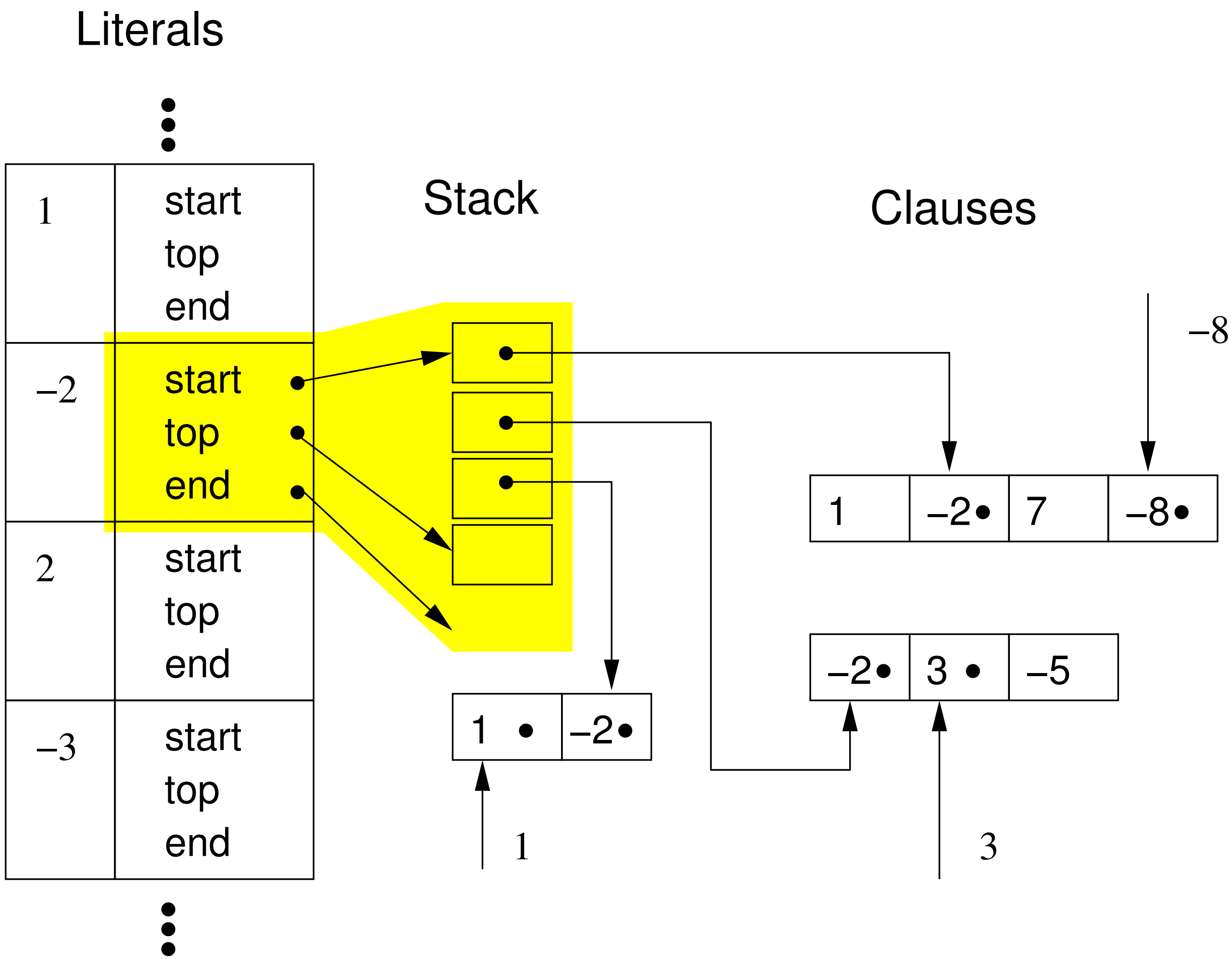
$(\overline{d} \vee \overline{g} \vee \overline{s})$

algorithm of Allen Van Gelder in SAT'09 produces regular input resolution proofs directly

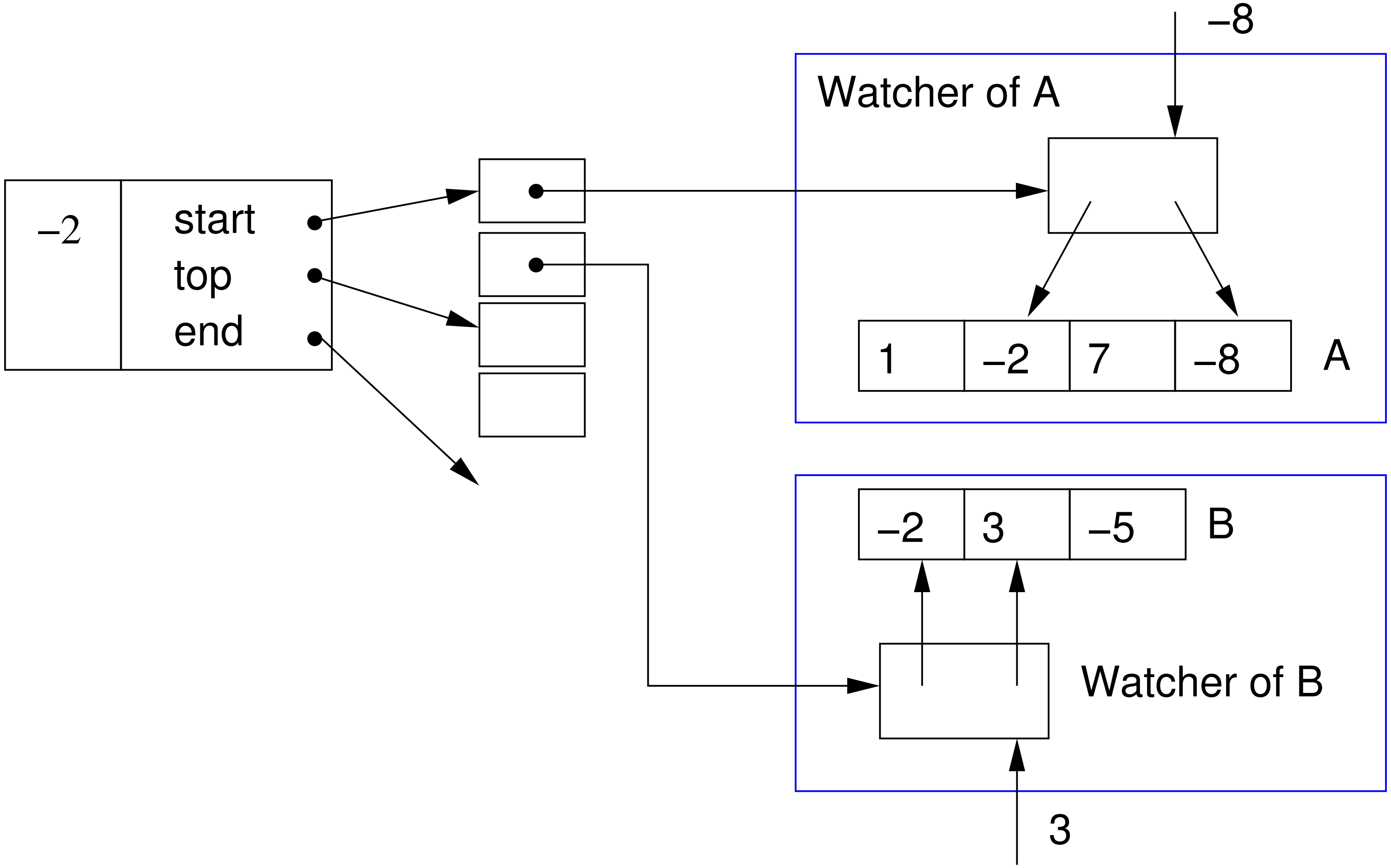
Two-Watched Literal Schemes

- original idea from SATO [ZhangStickel'00]
  - maintain the invariant: $\boxed{\text{always watch two non-false literals}}$
  - if a watched literal becomes *false* replace it
  - if no replacement can be found clause is either unit or empty
  - original version used *head* and *tail* pointers on Tries
- improved variant from Chaff [MoskewiczMadiganZhaoZhangMalik'01]
  - watch pointers can move arbitrarily — SATO: *head* forward, *trail* backward
  - no update needed during backtracking
- *one* watch is enough to ensure correctness — but looses *arc consistency*
- reduces *visiting* clauses by 10x, particularly useful for large and many learned clauses

ZChaff Occurrence Stacks

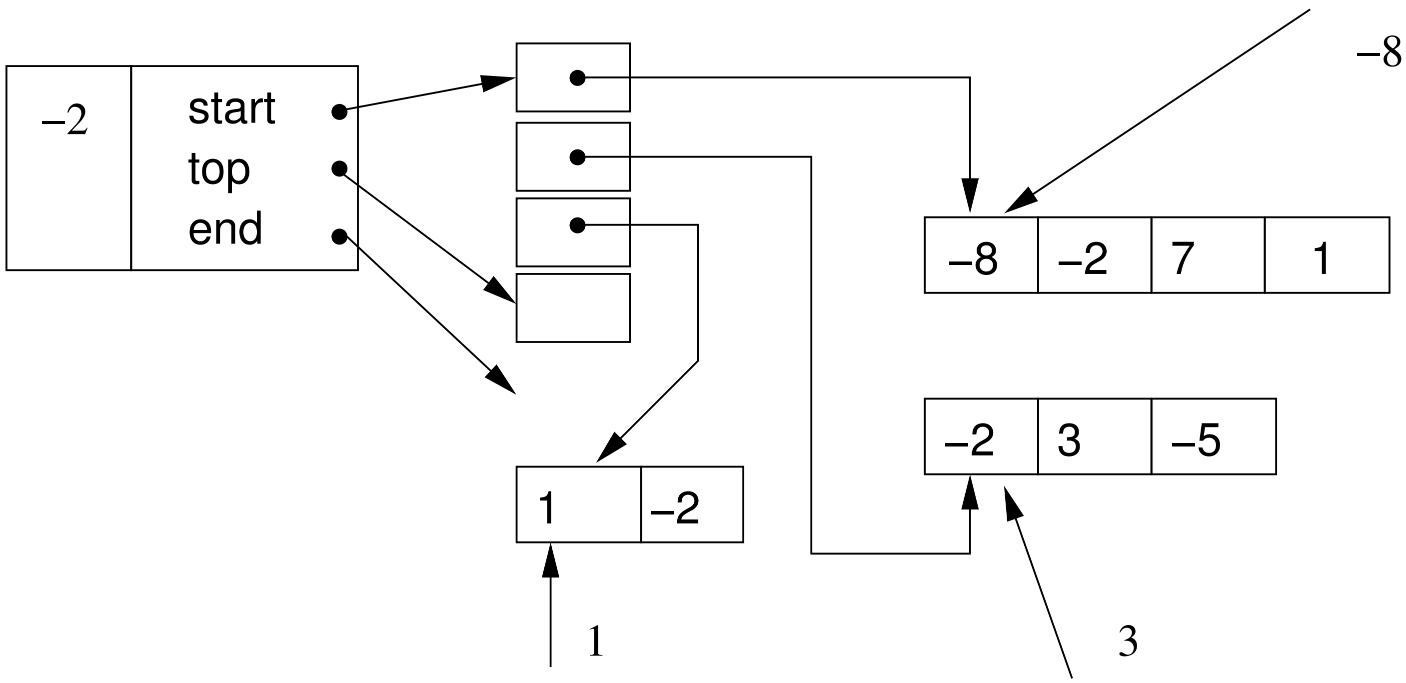


Limmat / FunEx Occurrence Stacks



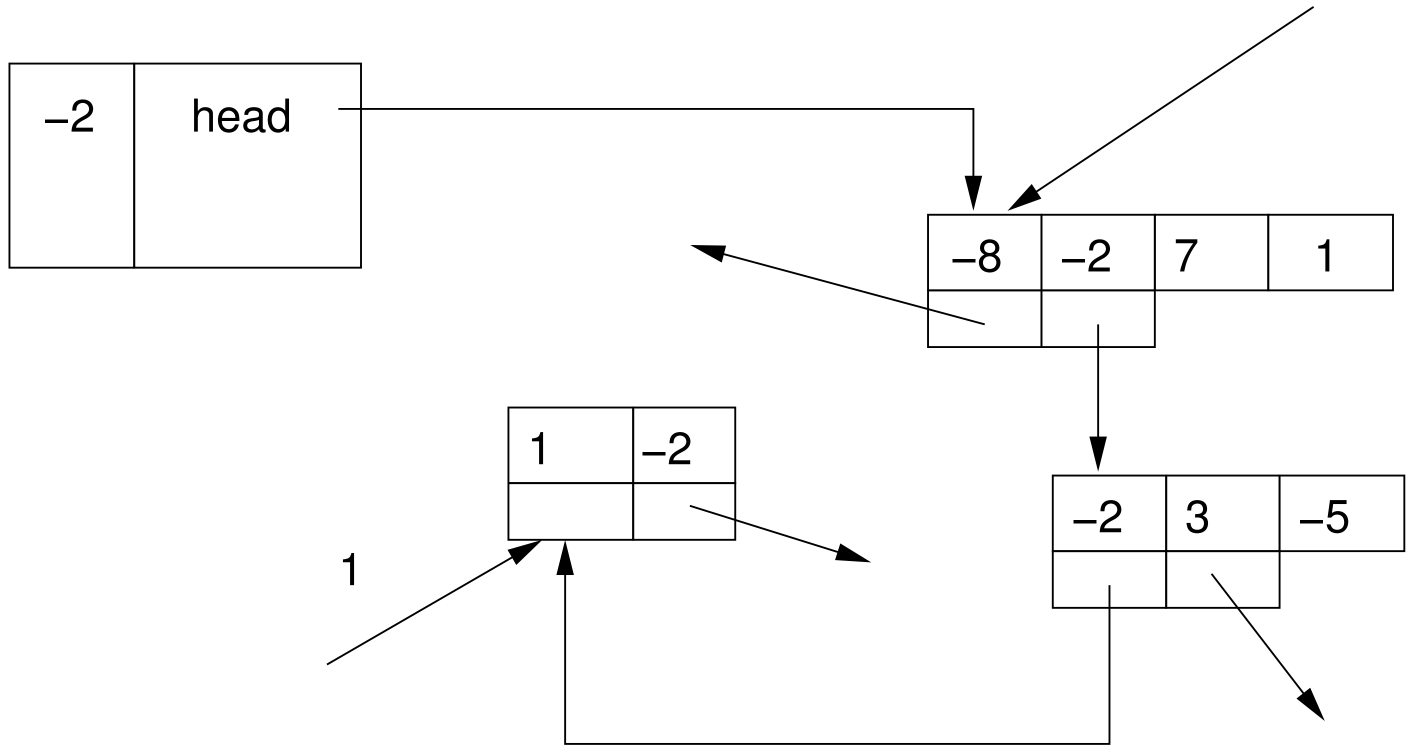
still seems to be best way for *real* sharing of clauses in multi-threaded solvers

CompSAT / MiniSAT Occurrence Stacks



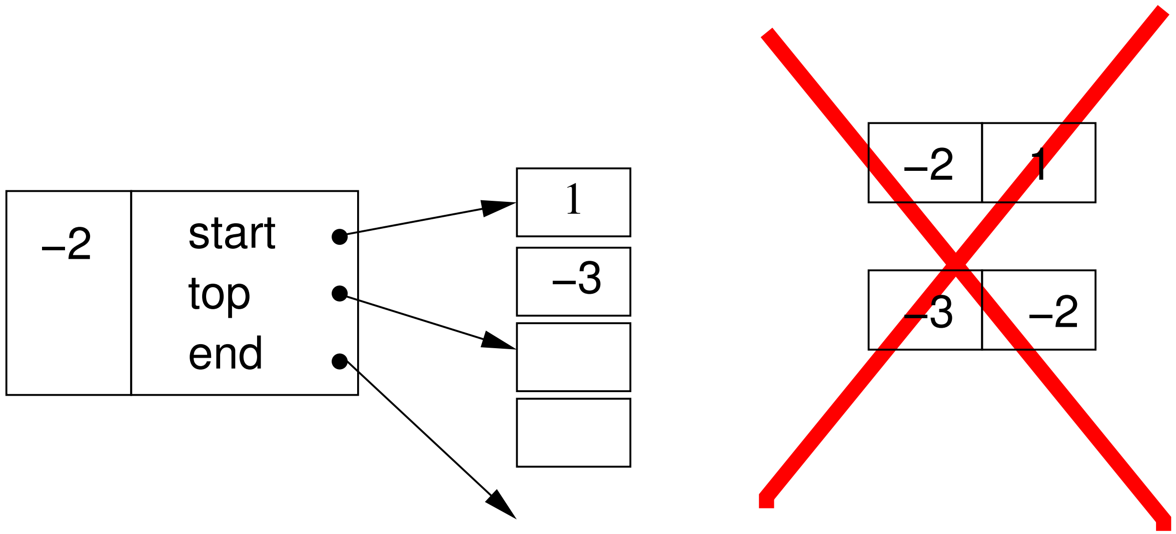
invariant: first two literals are watched

MChaff / PicoSAT Occurrence Lists



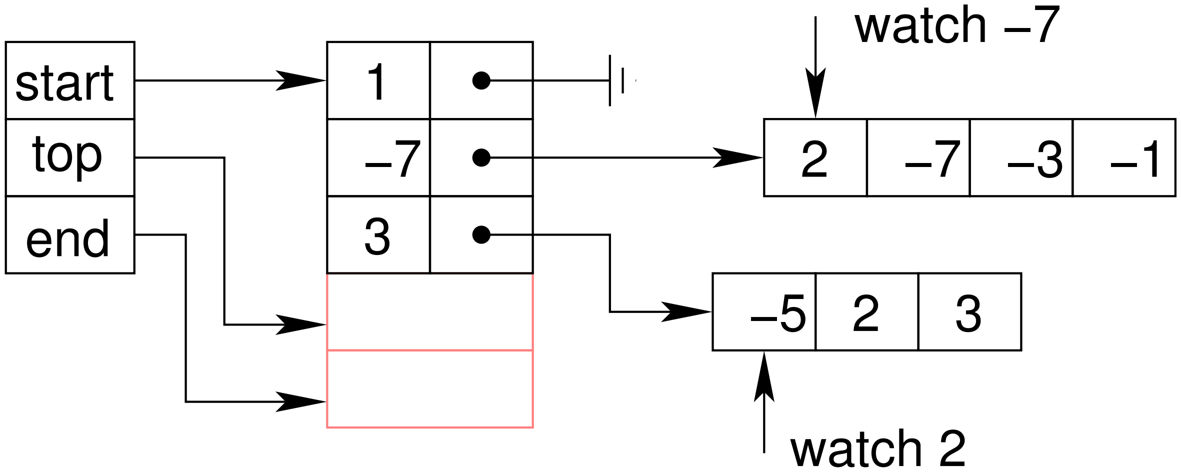
invariant: first two literals are watched

Additional Binary Clause Watcher Stack



Caching Potential Satisfied Literals (Blocking Literals)

ChuHarwoodStuckey'09



observation: often the *other* watched literal satisfies the clause

so cache this literals in watch list to avoid pointer dereference

for binary clause no need to store clause at all

can easily be adjusted for ternary clauses (with full occurrence lists)

LINGELING uses more compact pointer-less variant

Failed Literal Probing

we are still working on tracking down the origin before [Freeman'95] [LeBerre'01]

- key technique in look-ahead solvers such as Satz, OKSolver, March
  - failed literal probing at all search nodes
  - used to find the best decision variable and phase
- simple algorithm
  1. assume literal l, propagate (BCP), if this results in conflict, add unit clause $\neg l$
  2. continue with all literals l until *saturation* (nothing changes)
- quadratic to cubic complexity
  - BCP linear in the size of the formula — 1st linear factor
  - each variable needs to be tried — 2nd linear factor
  - and tried again if some unit has been derived — 3rd linear factor

Failed Literal Probing Extensions

- lifting
  - complete case split: literals implied in all cases become units
  - similar to Stålmark's method and Recursive Learning [PradhamKunz'94]
- asymmetric branching
  - assume all but one literal of a clause to be false
  - if BCP leads to conflict remove originally remaining unassigned literal
  - implemented for a long time in MiniSAT but switched off by default
- generalizations:
  - vivification [PietteHamadiSais ECAI'08]
  - distillation [JinSomenzi'05][HanSomenzi DAC'07] probably most general (+ tries)

Other Types of Learning

- similar to look-ahead heuristics: polynomially bounded search
  - can be applied recursively (however, is often too expensive)
- Stålmarck's Method
  - works on triplets (intermediate form of the Tseitin transformation):
    $x = (a \wedge b)$, $y = (c \vee d)$, $z = (e \oplus f)$ etc.
  - generalization of BCP to (in)equalities between variables
  - **test rule** splits on the two values of a variable
- Recursive Learning (Kunz & Pradhan)
  - (originally) works on circuit structure (derives implications)
  - splits on different ways to *justify* a certain variable value

Bounded Variable Elimination (BVE)

[DavisPutnam60][Biere SAT'04] [SubbarayanPradhan SAT'04] [EénBiere SAT'05]

- use DP to existentially quantify out variables as in [DavisPutnam60]

- only remove a variable if this does not add (too many) clauses

  - do not count tautological resolvents

  - detect units on-the-fly

- schedule removal attempts with a priority queue [Biere SAT'04] [EénBiere SAT'05]

  - variables ordered by the number of occurrences

- strengthen and remove subsumed clauses (on-the-fly)
  (SATeLite [EénBiere SAT'05] and Quantor [Biere SAT'04])

Fast (Self) Subsumption

- for each (new or strengthened) clause
  - traverse list of clauses of the least occuring literal in the clause
  - check whether traversed clauses are subsumed or
  - strengthen traversed clauses by self-subsumption [EénBiere SAT'05]
  - use Bloom Filters (as in "bit-state hashing"), aka signatures
- check old clauses being subsumed by new clause: backward (self) subsumption
  - new clause (self) subsumes existing clause
  - new clause smaller or equal in size
- check new clause to be subsumed by existing clauses forward (self) subsumption
  - can be made more efficient by one-watcher scheme [Zhang-SAT'05]

Blocked Clause Elimination (BCE)

one clause $C \in F$ with l *all* clauses in F with $\bar{l}$

$\bar{l} \vee \bar{a} \vee c$

fix a CNF F

$a \vee b \vee l$

$\bar{l} \vee \bar{b} \vee d$

all resolvents of C on l are tautological $\Rightarrow$ C can be removed

Proof assume assignment σ satisfies $F \backslash C$ but not C

can be extended to a satisfying assignment of F by flipping value of l

Definition A literal l in a clause C of a CNF F blocks C w.r.t. F if for every clause $C' \in F$ with $\bar{l} \in C'$, the resolvent $(C \setminus \{l\}) \cup (C' \setminus \{\bar{l}\})$ obtained from resolving C and C' on l is a tautology.

Definition [Blocked Clause] A clause is blocked if has a literal that blocks it.

Definition [Blocked Literal] A literal is blocked if it blocks a clause.

Example $(a \vee b) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee \boxed{c})$

only first clause is not blocked.

second clause contains two blocked literals: a and $\bar{c}$.

literal $\boxed{c}$ in the last clause is blocked.

after removing either $(a \vee \bar{b} \vee \bar{c})$ or $(\bar{a} \vee c)$, the clause $(a \vee b)$ becomes blocked
actually all clauses can be removed

JärvisaloBiereHeule'10 + JAR Article

COI Cone-of-Influence reduction

MIR Monontone-Input-Reduction

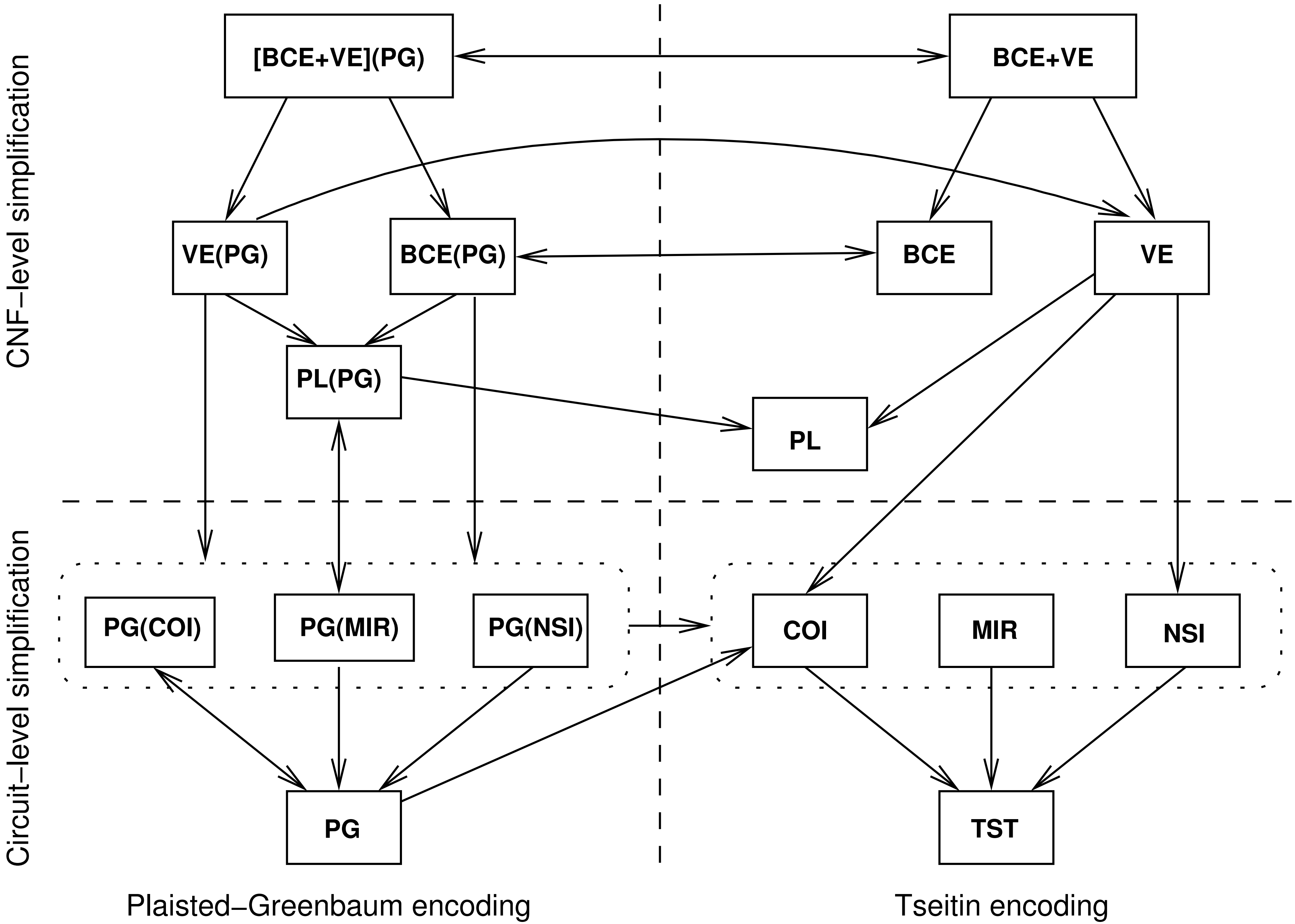
NSI Non-Shared Inputs reduction

PG Plaisted-Greenbaum polarity based encoding

TST standard Tseitin encoding

VE Variable-Elimination as in DP / Quantor / SATeLite

BCE Blocked-Clause-Elimination



CNF−level simplification
Circuit−level simplification
[BCE+VE](PG)
BCE+VE
VE(PG)
BCE(PG)
BCE
VE
PL(PG)
PL
PG(COI)
PG(MIR)
PG(NSI)
COI
MIR
NSI
PG
TST
Plaisted−Greenbaum encoding
Tseitin encoding

Inprocessing: Interleaving Preprocessing and Search

PrecoSAT [Biere'09], Lingeling [Biere'10], now also in CryptoMiniSAT (Mate Soos)

- preprocessing can be extremely beneficial
  - most SAT competition solvers use variable elimination (VE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple idea to benefit from full preprocessing without penalty
  - "preempt" preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time

equivalent literal substitution find strongly connected components in binary implication graph, replace equivalent literals by representatives

boolean ring reasoning extract XORs, then Gaussian elimination etc.

hyper-binary resolution focus on producing binary resolvents

hidden/asymmetric tautology elimination discover redundant clauses through probing

covered clause elimination use covered literals in probing for redundant clauses

unhiding randomized algorithm (one phase linear) for clause removal and strengthening

Benefits of Inprocessing

- allows to use *costly* preprocessors
  - without increasing run-time “much” in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and distillation — even VE can be costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
  - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding “bad” implementation though . . .
- . . . and hard(er) to get right! “Inprocessing Rules” [JärvisaloHeuleBiere’12]

$$\frac{\varphi\,[\,\rho \wedge C\,]\,\sigma}{\varphi \wedge C\,[\,\rho\,]\,\sigma}$$

STRENGTHEN

$$\frac{\varphi\,[\,\rho \wedge C\,]\,\sigma}{\varphi\,[\,\rho\,]\,\sigma}$$

FORGET

$$\frac{\varphi\,[\,\rho\,]\,\sigma}{\varphi\,[\,\rho \wedge C\,]\,\sigma}\;\mathcal{L}$$

LEARN

$$\frac{\varphi \wedge C\,[\,\rho\,]\,\sigma}{\varphi\,[\,\rho \wedge C\,]\,\sigma, l{:}C}\;\mathcal{W}$$

WEAKEN

$\mathcal{L}$ is that $\boxed{\varphi \wedge \rho}$ and $\boxed{\varphi \wedge \rho \wedge C}$ are satisfiability-equivalent.

$\mathcal{W}$ is that $\boxed{\varphi}$ and $\boxed{\varphi \wedge C}$ are satisfiability-equivalent.

RAT – Resolution Asymmetric Tautology

idea: “resolution look-ahead”

Clause R *asymmetric tautology* (AT) w.r.t. G iff $G \wedge \neg C$ refuted by BCP.

Given clause $C \in F$, $l \in C$.

Assume all resolvents R of C on l with clauses in F are AT w.r.t. $F \backslash \{C\}$.

Then C is called *resolution asymmetric tautology* (RAT) w.r.t. F on l.

In this case F is satisfiability equivalent to $F \backslash \{C\}$.

Inprocessing Rules with RAT simulate all techniques in current SAT solvers

- application level parallelism
  - run multiple “properties” at the same time
  - run multiple “engines” at the same time (streaming)
- portfolio solving
  - predict best solver through machine learning techniques SATzilla
  - run multiple solvers in parallel or sequentially (with/without “sharing”)
    ManySAT, Plingeling, ppfolio ...
- split search space
  - guiding path principle [ZhangBonacinaHsiang’96]
  - cube & conquer [HeuleKullmanWieringaBiere’11]
- low-level parallelsim: parellize BCP (threads, FPGA, GPU, ...) P complete

- use Look-Ahead at the top of the search tree, CDLC at the bottom
  - Look-Ahead solvers provide "good" global decisions but are slow
  - CDCL solvers are extremely fast based on local heuristics
- **when to switch from Look-Ahead to CDCL?**
  - limit decision height of Look-Ahead solver, e.g. 20 maximum tree height
  - avoid having too many branches closed by (slow) Look-Ahead
  - Concurrent Cube & Conquer runs CDCL and Look-Ahead in parallel
- open branches = cubes $\Rightarrow$ solve in parallel
- solves hard instances, which none of the other approaches can